2

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD-A206 113

THESIS

DESIGN AND IMPLEMENTATION OF AN EXPERT USER INTERFACE FOR THE COMPUTER AIDED PROTOTYPING SYSTEM

by

Henry G. Raum

December 1988

Thesis Advisor: Luqi

Approved for public release; distribution is unlimited

Unclassified

Security Classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification Unclassified	1b Restrictive Markings
2a Security Classification Authority	3 Distribution Availability of Report
2b Declassification/Downgrading Schedule	Approved for public release; distribution is unlimited.
4 Performing Organization Report Number(s)	5 Monitoring Organization Report Number(s)
6a Name of Performing Organization Naval Postgraduate School / 6b Office Symbol (If Applicable) 37	7a Name of Monitoring Organization Naval Postgraduate School
6c Address (city, state, and ZIP code) Monterey, CA 93943-5000	7b Address (city, state, and ZIP code) Monterey, CA 93943-5000
8a Name of Funding/Sponsoring Organization / 8b Office Symbol (If Applicable)	9 Procurement Instrument Identification Number
8c Address (city, state, and ZIP code)	10 Source of Funding Numbers: Program Element Number / Project No / Task No / Work Unit Accession No

11 Title (Include Security Classification) Design and Implementation of an Expert User Interface for the Computer Aided Prototyping System

12 Personal Author(s) Henry G. Raum

13a Type of Report Master's Thesis	13b Time Covered From To	14 Date of Report (year, month,day) December 1988	15 Page Count 92

16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms (continue on reverse if necessary and identify by block number)
Field	Group	Subgroup	User Interface, Expert System, Rapid Prototyping, Computer Aided Software Engineering (CASE), Ada, Real-Time, Embedded Systems.

19 Abstract (continue on reverse if necessary and identify by block number

This thesis builds on previous work done in the development of the Computer Aided Prototyping System (CAPS) and the Prototype System Description Language (PSDL). The increases in the size and complexity of software projects have caused system designers to reevaluate traditional software engineering methodologies. Rapid prototyping is a method that allows the validation of system requirements and design early in the development cycle. The need for this type of tool is particularly critical in the development of real-time embedded systems. CAPS is one such system.

CAPS is a complex system that consists of many individual software tools. An expert user interface that guides the software designers through the development and execution of a prototype is described in this thesis.

20 Distribution/Availability of Abstract [✓ unclassified/unlimited] same as report DTIC users	21 Abstract Security Classification Unclassified	
22a Name of Responsible Individual Luqi	22b Telephone (Include Area code) (408) 646-2735	22c Office Symbol 52Lq

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted security classification of this page

All other editions are obsolete Unclassified

DESIGN AND IMPLEMENTATION OF AN EXPERT USER INTERFACE FOR THE COMPUTER AIDED PROTOTYPING SYSTEM

by

Henry G. Raum
Captain, United States Marine Corps
B.S., United States Naval Academy, 1980

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 1988

Author: [illegible]
Henry G. Raum

Approved by: [illegible]
Luqi, Thesis Advisor

[illegible]
John M. Yurchak, Second Reader

[illegible]
Robert McGhee, Chairman
Department of Computer Science

[illegible]
Kneale T. Marshall,
Dean of Information and Policy Sciences

ABSTRACT

This thesis builds on previous work done in the development of the Computer Aided Prototyping System (CAPS) and the Prototype System Description Language (PSDL). The increases in the size and complexity of software projects have caused system designers to reevaluate traditional software engineering methodologies. Rapid prototyping is a method that allows the validation of system requirements and design early in the development cycle. The need for this type of tool is particularly critical in the development of real-time embedded systems. CAPS is one such system.

CAPS is a complex system that consists of many individual software tools. An expert user interface that guides the software designers through the development and execution of a prototype is described in this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

CAPS	Computer Aided Prototyping System
DDB	Design Database
DFD	Data Flow Diagram
DOD	Department of Defense
DON	Department of the Navy
DS	Dynamic Scheduler
ESS	Execution Support System
GE	Graphic Editor
MCP	Minimum Calling Period
MET	Maximum Execution Time
MRT	Maximum Response Time
PSDL	Prototype System Description Language
SB	Software Base
SDE	Syntax Directed Editor
TR	Translator
UI	User Interface

I. INTRODUCTION

Over the last twenty years, advancements in computer hardware have far exceeded those in software development. This "software crisis" must be resolved if improved performance of systems is expected [Ref. 7:p. 11]. The most significant problem is that the users of the software systems often do not understand software engineering and the software engineers do not understand the needs of the users. As a result, poor requirements analysis often leads to an improper design. In the traditional software engineering paradigm testing of the system is done only after coding is complete. This results in much wasted effort as improper specifications are coded.

An alternate approach to software engineering is prototyping. In this method a model of the eventual program is quickly constructed and tested. The goal is to evaluate the design of the system and make modifications, if necessary, before the coding is started. [Ref. 2:p. 69]

One system that implements the prototyping methodology is the Computer Aided Prototyping System (CAPS). This system constructs a prototype from specifications described in the Prototyping System Description Language (PSDL) and generates an executable model of the real-time system. This model is actually an Ada®[1] program that can be executed and modified until the prototype is performing correctly. [Ref. 3]

[1] Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

CAPS is applicable to the Department of the Navy (DON) and Department of Defense (DOD) because it generates an Ada® prototype. With the requirement that all embedded systems in DOD be implemented in Ada®, an Ada® prototype that demonstrates use of the language constructs makes CAPS more attractive.

The users of this system may not be familiar with PSDL or CAPS and may not even have any software engineering expertise. For this reason, an intelligent user interface that guides the user through the production of the prototype is required. This thesis will describe the design and implementation of such a user interface.

The remainder of this chapter will describe the traditional and rapid prototyping software engineering methods and the CAPS system. This will include a description of the elements and processes of CAPS and an overview of PSDL.

A. DESCRIPTION OF SOFTWARE ENGINEERING METHODOLOGIES.

1. Traditional Software Engineering Paradigm

The traditional software engineering paradigm, often called the "waterfall model", is a systematic, sequential approach [Ref. 2:p. 13]. This model begins with requirements analysis and continues with functional specification, design, coding and testing. Requirements analysis defines the scope of the system, and the environment it will operate in, while the functional specification describes interfaces to the proposed system and the functions of that system. The design stage includes the decomposition and detailed design. The data structures, software architecture and procedural details are key issues at this stage. The coding stage is where the designed system is written into a form usable by the computer, normally a high level programming language. Finally, testing is done. These tests not only show that the program meets the requirements and specifications, but determines if these requirements produce the desired results. [Ref. 4]

The fact that coding and testing occurs so late in the process has brought much criticism of the "waterfall model". It is difficult for the customer to furnish complete requirements before the process begins. In large software systems this can lead to a disaster if a major oversight in the requirement analysis phase is not discovered until testing. [Ref. 2:p. 15]

2. Prototyping in Software Engineering.

In response to the problems in the "waterfall model" a new approach to software engineering, called prototyping, has been devised. Prototyping is a method that is well suited to the iterative nature of the development of many software systems. When the customers requirements cannot be completely determined or there are questions about the suitability of proposed algorithims or what the human interface should be, prototyping allows a model to be built which can be tested and modified. After testing the prototype, a new set of improved specifications can be used in the coding phase.

Figure 1 illustrates how, in this method, testing and refinement of requirements is done before the actual product is engineered. Significant time and cost savings can be realized because code is not being written for incomplete and erroneous specifications. The complex and uncertain timing requirements of real-time embedded systems makes them good candidates for prototyping. [Ref. 2]
The problem with prototyping is the cost of developing the prototype itself. Paper prototypes are easy to construct, but do not show the dynamic nature of the system. Working prototypes can show the feel of the program, but are limited to a subset of the system. In the Computer Aided Prototyping System the entire system is

modeled. The time required to build this prototype is greatly reduced through the use of syntax directed and graphic editors, reusable Ada® modules and a self-contained execution and debugging system. CAPS attempts to validate the timing aspects of real-time embedded systems by constructing a prototype of the entire system, executing that prototype and measuring its performance. As part of the software engineering effort, CAPS falls between the requirements analysis and coding phases.

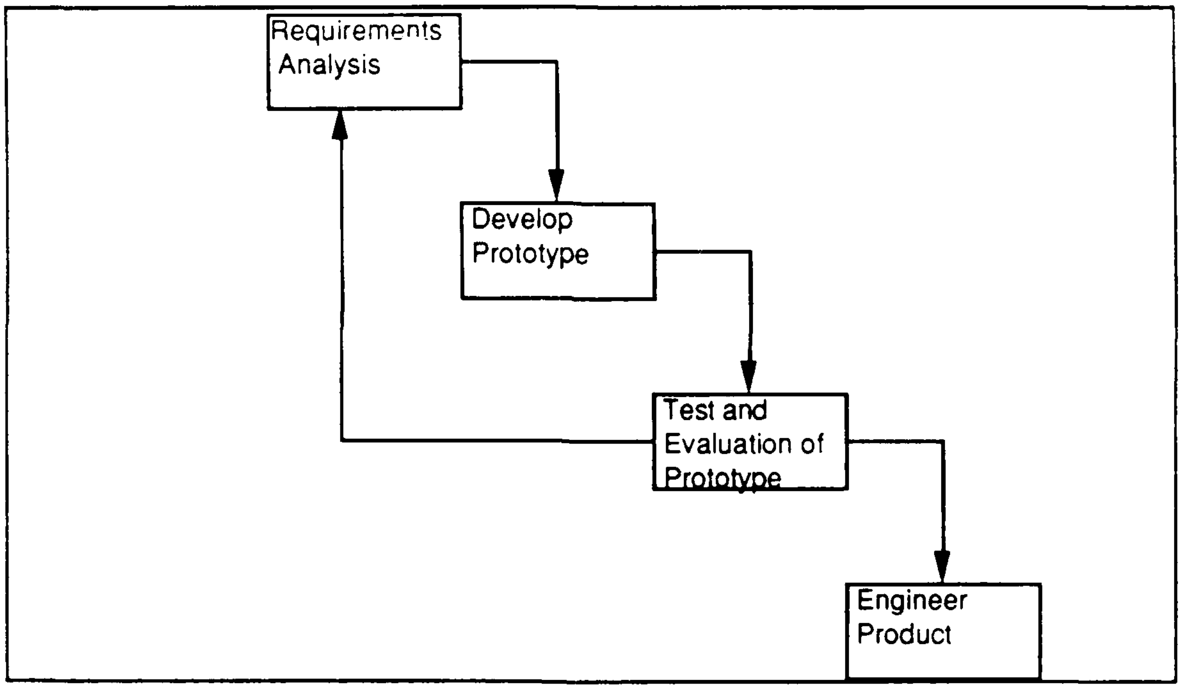


Figure 1. The Prototyping Method

B. OBJECTIVES

This thesis describes the design and implementation of an expert user interface for CAPS. The primary objective is to define the proper use of database, graphic, and operating systems technology in the design of the interface. Secondly, this thesis will explore the use of expert system technology to produce an interface that can free the user from many of the details of the system operation. In this way, an error free prototype can be produced quickly and easily.

C. ORGANIZATION

Chapter II outlines the background research and includes a description of CAPS, the Prototype System Description Language (PSDL) and the principles of the user interface in software systems.

Chapter III describes a design for the user interface to include the interfaces between CAPS and the user as well as those between the various elements of the system. This chapter includes a users manual and recommendations for future implementation.

Chapter IV outlines the implementation of the design, both what has been done and what is recommended for future implementation.

Chapter V contains conclusions and recommendations of this research.

II. BACKGROUND

The Computer Aided Prototyping System is a complex collection of software tools that enable the user to produce executable prototypes of large real-time embedded systems. This chapter outlines the components of CAPS and PSDL in detail. Finally the principles of the user interface are investigated.

A. DESCRIPTION OF CAPS

1. Architecture of CAPS

CAPS consists of many software tools that each have a part in the production of the prototype. The position of each component in the overall system is illustrated in Figure 2. A brief description of the components follows.

a. The User Interface Module.

This module consists of two editors, a syntax directed editor and a graphical editor. Together these editors produce the PSDL program that will become the prototype.

(1) The Syntax Directed Editor is an editor that knows the key words and proper syntax of PSDL. It insures grammatically correct PSDL as it guides the user through the production of the program.

(2) The Graphic Editor is an editor that describes the decomposition of PSDL components by the use of enhanced data flow diagrams. It is in these diagrams that the user "creates" the program.

(3) The Design Database is the structure that stores the elements of the system under construction. This database is organized to handle the hierarchical

nature of the top-down development of the system. The design database also has the ability to keep a history of the prototype when changes are made. [Ref. 4:p. 29]

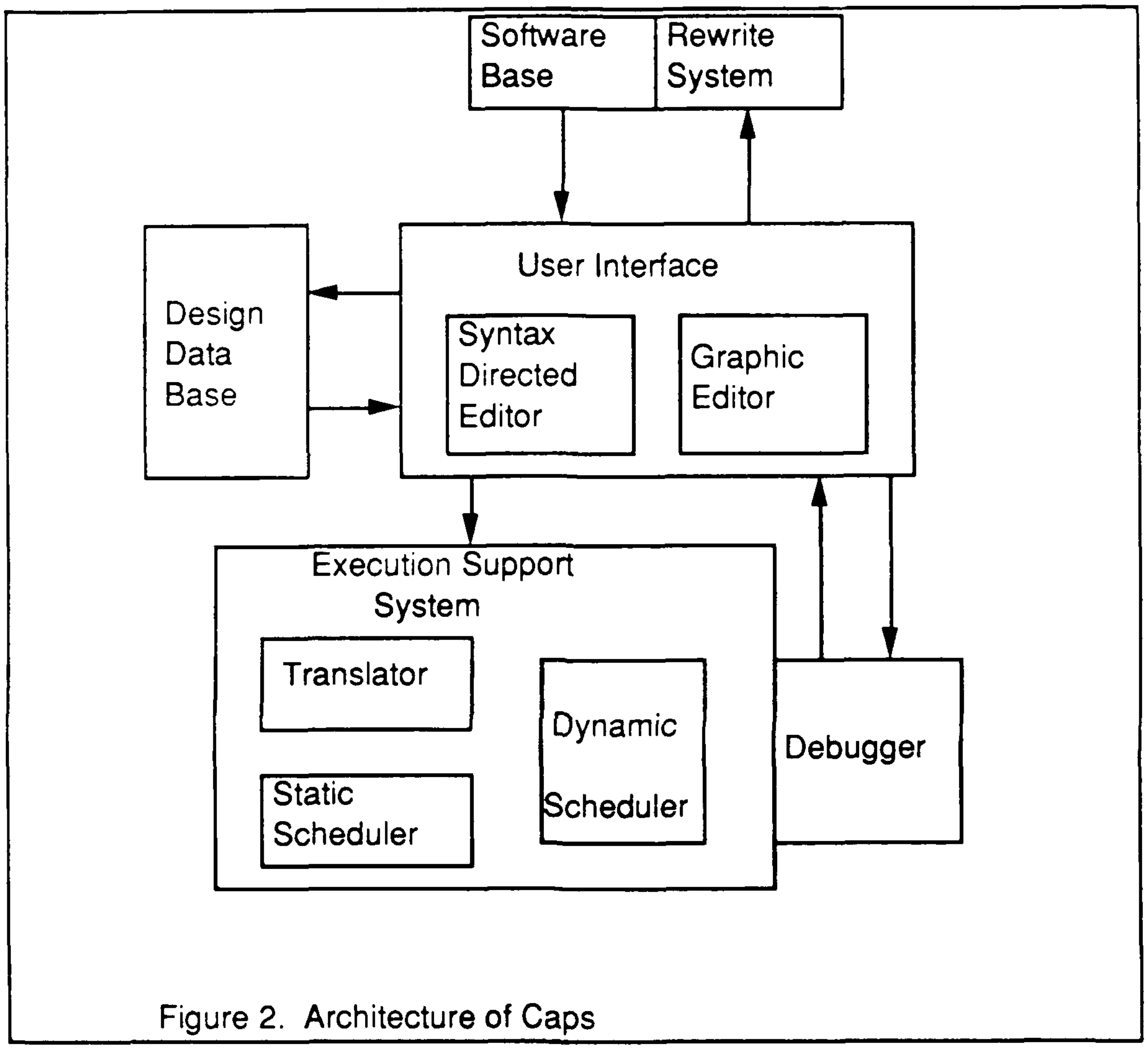


Figure 2. The Architecture of CAPS

b. The Software Base and Rewrite System

Together the software base and rewrite system give CAPS its ability to retrieve reusable Ada® components for construction of the prototype.

(1) The Rewrite System. The purpose of the rewrite system is to provide a method for mapping all equivalent specifications to one common form. This is called the normalized form.

(2) The software base is a database of reusable Ada® components that are indexed and searched for based on PSDL specifications. This will provide an Ada® module that fulfills the desired function. In the production of a rapid prototype there is no time for browsing a software library by name, as in a yellow pages type of index. [Ref. 5:pp. 68-69]

c. *The Execution Support System*

The Execution Support System (ESS) consists of a translator, a static scheduler, a dynamic scheduler and a debugger. While the purpose of the rest of CAPS is to produce a PSDL program, the function of the ESS is to translate the PSDL program into a executable Ada® prototype that tests the real-time aspects of the designed system.

(1) Translator. The translator starts with the Ada® components retrieved in the software base and adds to them the control constants described in PSDL to produce an Ada® module that can be scheduled.

(2) The Static Scheduler. The static scheduler links all of the time critical components together in an executable schedule that can demonstrate the real-time aspects of the system.

(3) The Dynamic Scheduler. The dynamic scheduler adds the non-time critical components to the system to produce an overall schedule.

(4) The Debugger. The debugger provides an interface between the user and the executing prototype. The user may be asked to correct a problem in the prototype or be presented with statistics on the operation of the prototype.

2. Description of PSDL

The Prototype System Description Language is an executable prototyping language that is used at the design level. The user can define the specifications in this language and these specifications are used to search the software base of reusable Ada® components. If a suitable module is not found the PSDL operator is then decomposed by drawing an enhanced data flow diagram that includes lower level operators, the data streams between these operators, as well as timing and control constraints. [Ref. 5:p. 68]

a. Elements of PSDL

(1) Operators. Operators are the basic component in PSDL. They can represent functions or state machines. These operators can be triggered by the arrival of some input (sporadic) or at fixed time intervals (periodic). When triggered the operator will fire (produce output) based on input values and the value of the internal state variable in the case of the state machine. [Ref. 6:pp. 12-13]

Operators are called atomic if they can be found in the software base, otherwise they are called composite and must be decomposed with a data flow diagram. Figure 3 shows such a decomposition. Data streams w and z are the respective input and output of the composite operator and s is the state variable. A, B, and C are newly created operators that may be atomic or composite themselves. [Ref. 5]

(2) Data Streams. A data stream represents the flow of data between two operators. This communication can be in the form of a data flow stream or a sampled stream. A data flow stream can be thought of as a FIFO queue. In this way a data flow stream is never lost and is always acted on in the order of arrival. A

sampled stream can be thought of as a single memory cell. This data can be used many times or written over before use, depending on the rate of its input and use. [Ref. 3]

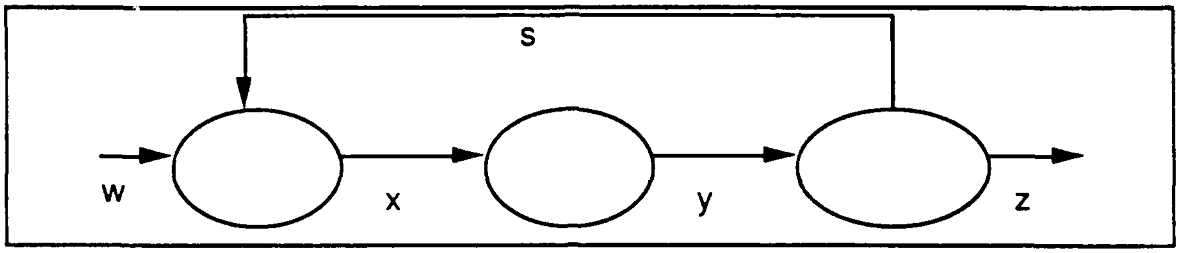


Figure 3. Graphic Decomposition

Data flow streams must be used when each piece of data represents a unique transaction. A sampled stream can simulate a sensor that is only interested in a current parameter such as temperature. In Figure 3, the data streams are w,x,y,and z, with x and y being the new data streams created in the decomposition.

(3) Timing and Control Constraints. The real-time nature of the prototypes requires timing and non-procedural control constraints in PSDL. Each time critical operator has a Maximum Execution Time, which is the maximum time in which that operator can complete execution after it fires. Control Constraints give conditional requirements on the firing of operators.

b. Example of PSDL.

To illustrate the PSDL program the example of the Hyperthermia system that was defined in the conceptual research for CAPS [Ref. 3] will be used. This example illustrates a dynamic real-time environment that is easy to understand.

> One approach to combating cancer is to destroy tumorous cells selectively with heat. One way to do this is with a hyperthermia system, which uses a microwave generator connected to a fine tuner and matching control system to produce and deliver controlled, local heating directly to the

tumors. A computerized control system adjusts power output automatically to maintain the temperature in accordance with the treatment plan.

The hyperthermia system has four subsystems: a computer system, an operator's panel, a microwave generator, and a temperature sensor. The critical subsystem is the software that receives input from the temperature sensor and produces control commands to operate the whole system. The software controls the rest of the system, which is typical of real-time embedded systems. [Ref. 4:p. 30]

(1) PSDL Specification. The operator is described in PSDL as shown in Figure 4. The Specification includes the operator name, an input list, an output list, a state list (if operator is a state machine), optional exception declarations and timing information. This specification is used to search the software base. If no match is found this composite operator is decomposed in the implementation part.

```
OPERATOR Brain_tumor_treatment_system
SPECIFICATION
 INPUT patient_chart: medical_history,
     treatment_switch : boolean
 OUTPUT treatment_finished: boolean
 STATES temperature: real INITIALLY 37.0
DESCRIPTION {This is an English description of the operator specification}
END
```

Figure 4. PSDL Operator Specification

Timing constraints may be added to a specification in the following ways:

- Maximum Execution Time (MET) which places a maximum time on the execution of an operator from initiation to completion.
- Maximum Response Time (MRT) has slightly different meanings for sporadic and periodic operators. It is the time between the start of the period and the moment of the last output of an operator for periodic operators and is the time between arrival of new data and the moment of output for sporadic operators.
- Minimum Calling Period (MCP) is required for any sporadic operator with an MRT and is the minimum time from the arrival of one set of data to the arrival of the next. [Ref. 6:pp. 20-21]

(2) PSDL Implementation. The implementation part of PSDL consists of the graph (data flow diagram), a list of new data streams and control constraints for the newly created operators. The data flow diagram shown in Figure 3 would appear in PSDL as the link statements shown after the keyword GRAPH in Figure 5. A link statement is of the form:

data_stream.from_operator:met-->to_operator [Ref. 7:p. 26] .

The from operator, in the case of an input, and the to operator, in the case of an output, will be represented by the keyword EXTERNAL. The maximum execution time of the operator, and the colon, are omitted for non-time critical operators. The Data Stream part lists the new internal data streams named in the graphical decomposition.

```
IMPLEMENTATION
GRAPH temperature.EXTERNAL -> hyperthermia_system
    patient_chart.EXTERNAL -> hyperthermia_system
    treatment_switch.EXTERNAL -> hyperthermia_system
    treatment_power.hyperthermia_system -> simulated_patient
    treatment_finished.hyperthermia_system -> EXTERNAL
    temperature.simulated_patient -> hyperthermia_system
DATA STREAM treatment_power : real
CONTROL CONSTRAINTS
  OPERATOR hyperthermia_system
    PERIOD 200 ms BY REQUIREMENTS shutdown
  OPERATOR simulated_patient
    PERIOD 200 ms
DESCRIPTION {some text about it}
END
```

Figure 5. PSDL Operator Implementation

The Implementation section is completed with the Control Constraint part of PSDL. These control constraints include data triggers, periods, conditionals, timers and exceptions. This information is used to define interconnections between operators.

The most common control method is the period and data trigger. The period is shown in Figure 5 and indicates the synchronous timing of the operator. The data trigger is used to indicate control from arriving data. Two examples of data triggers are:

OPERATOR A TRIGGERED BY ALL x,y,z.

OPERATOR B TRIGGERED BY SOME v,w.

The by ALL trigger causes the operator to fire when all three inputs are present, while the by SOME trigger fires the operator when any value of v or w arrives. [Ref. 3:p. 17]

Conditional Constraints add an IF predicate to the data trigger. This boolean condition must be satisfied before the operator will fire. Examples of the conditional constraint are:

OPERATOR A TRIGGERED if b<10.

OPERATOR X OUTPUT y if z: critical.

A timer is used as an internal state that can be started, stopped, reset, and its current time read by the operator to allow that operator to do its own timing control if necessary. It is also possible to raise exceptions in PSDL. These exceptions can be user or system defined and are raised when unusual or catastrophic conditions are encountered. The exception is handled by an exception handler at the highest level of the program.

Data Streams utilize the operator as both its specification and implementation section as shown in Figure 6. Therefore, only the construction of operators needs to be discussed.

```
TYPE medical_history
SPECIFICATION
 OPERATOR get_tumor_diameter
  SPECIFICATION
      INPUT patient_chart: medical_history,
      tumor_location: string
   OUTPUT diameter: real
   EXCEPTIONS no_tumor
   MAXIMUM EXECUTION TIME 5 ms
   DESCRIPTION
   {This is an English description of the operator}
 END

IMPLEMENTATION
 tuple[tumor_desc: map[from: string, to: real], ...]
 OPERATOR
  IMPLEMENTATION
   GRAPH
    patient_chart.EXTERNAL-->tuple.get_tumor_desc
    tumor_location.EXTERNAL-->map.fetch
    diameter.map.fetch:4ms-->EXTERNAL
    td.tuple.get_tumor_desc:1ms-->map.fetch

 DATA STREAM td: tumor_desc

 CONTROL CONSTRAINTS
  OPERATOR map.fetch
   EXCEPTIONS no_tumor IF not(map.has(tumor_location, td))

 END
```

Figure 6. PSDL Data Type Specification and Implementation

3. The CAPS Process

There are four major elements in the CAPS process: prototype design, construction, execution and debugging/modification. The initial step, called prototype design, actually takes place outside CAPS. The purpose of CAPS is not to design a system, but rather to test and validate that design. Construction takes place in the user interface portion of the system, while execution takes place in the execution support system. Debugging and modification requires action in both the user interface and execution support systems.

a. Prototype Design.

The design of the prototype starts with an analysis of the problem and a decision as to what part or parts of the proposed system are to be prototyped. Then requirements for the prototype are generated. These requirements are usually written in English, but could be specified in a more formal notation. Example requirements given in English taken from the hyperthermia example are:

- Shutdown. Microwave power must drop to zero within 300 ms of turning off the treatment switch.
- Temperature Tolerance. After the system stabilizes, the temperature must be kept between 42.4°C and 42.6°C
- Maximum Temperature. The temperature must never exceed 42.6°C.
- Startup Time. The system must stabilize within 5 minutes of turning on the treatment switch.
- Treatment Time. The system must shut down automatically when the temperature has been above 42.4°C for 45 minutes. [Ref. 3:pp. 28-29]

The requirements are refined by asking the customer questions to determine exactly what the requirements mean and if they are complete. With the completion of the preliminary design, the construction of the prototype may begin.

b. Prototype Construction.

Construction of the prototype involves the use of the syntax directed editor, graphic editor, design database, rewrite system, software base and the sequence control of the CAPS user interface. This process is illustrated in Figure 7 and is described below.

To start the system, the PSDL specification of the operator that represents the entire prototype is produced in the syntax directed editor. After this specification has been normalized in the rewrite system a search of the software base of reusable Ada® modules is performed. This search by specifications produces no match, one match, or many matches. In the case of a single match, that Ada® component is used as the implementation part of the operator. A multiple match must be resolved and then the single resolved component is used. When there is no match, the operator is a candidate for decomposition.

Decomposition of an operator takes place in the graphic editor with the production of the enhanced data flow diagram. In this diagram the editor creates children operators of the current operator. In effect a multiway tree is produced that is rooted at the original operator and has atomic operators as its frontier. If the operator is a candidate for decomposition, but the designer feels the operator is too simple for further decomposition, it may be simply coded in Ada®. At this point, this operator would be atomic with this newly written code as its implementation.The design database keeps track of all the nodes created in the decomposition.

The primary responsibility of this database is to provide a storage structure for the PSDL components of each operator in a way that maintains the hierarchical nature of the prototype construction.

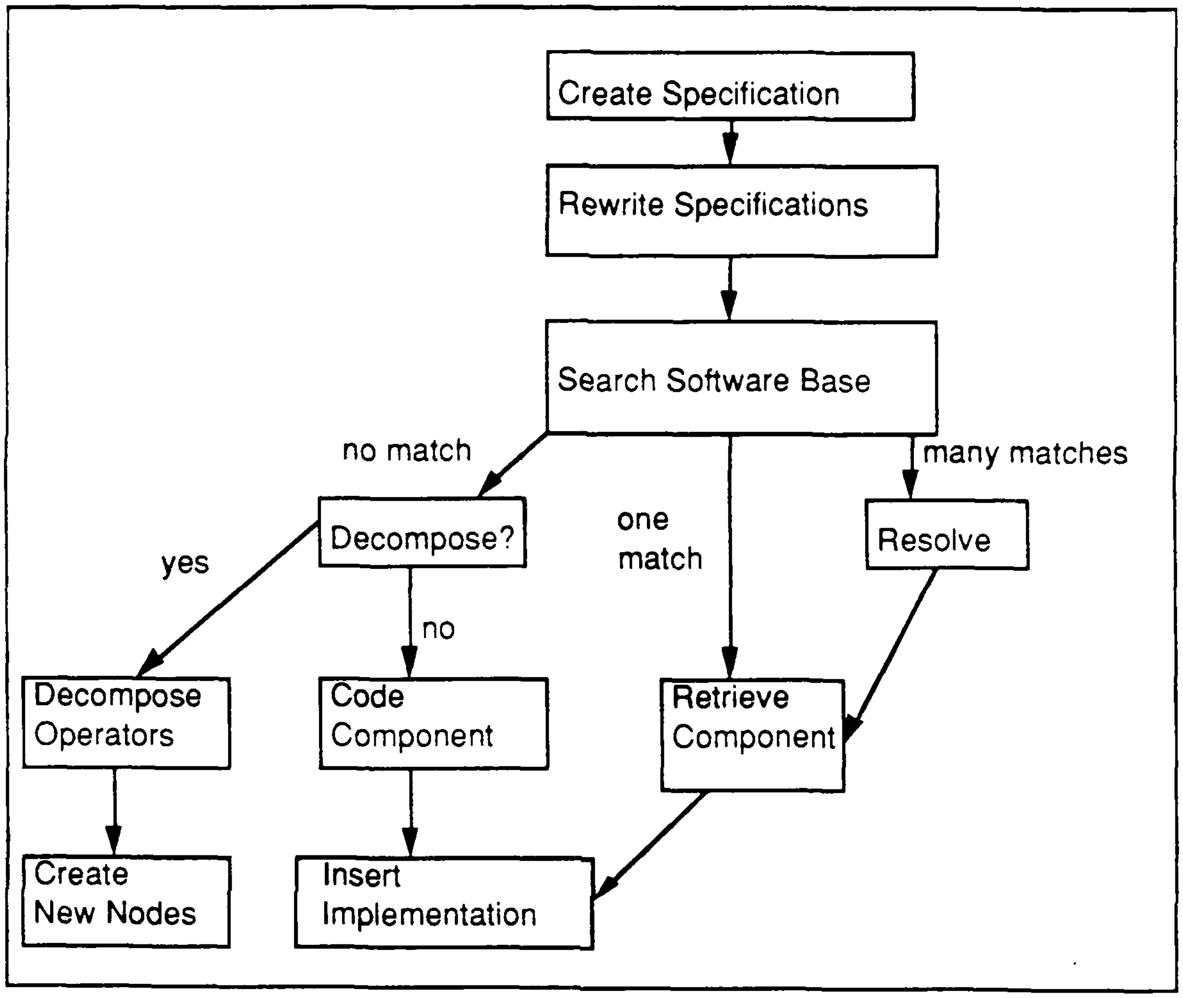


Figure 7. PSDL Construction

The user interface controls the flow of data and the use of the various tools in the system. This control includes calling the proper tool at any given moment and the repetition of the constructive process until all leaf operators in the design database tree are atomic.

c. *Prototype Execution.*

Execution of the prototype occurs in the execution support system (ESS), through the use of the translator, static scheduler, dynamic scheduler and the debugger as illustrated in Figure 8. The input to the ESS is the PSDL program that was previously constructed.

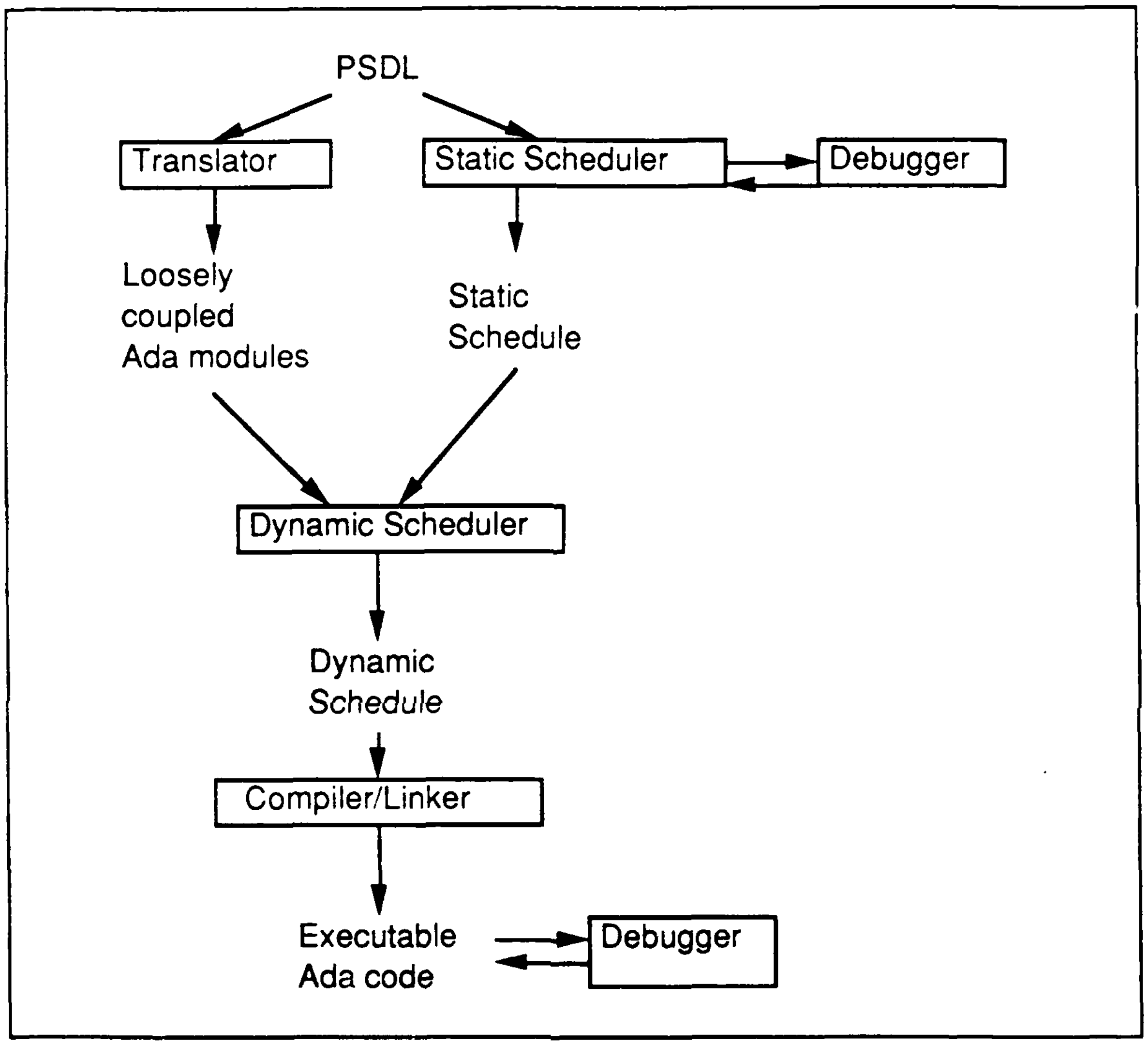


Figure 8. Execution Support System

The first component in the ESS is the translator, which translates PSDL into Ada® source code. The atomic operators are tempered by the control constraints of the composite operators and executable Ada® packages are produced. This collection of modules is used by the static and dynamic schedulers in the execution of the prototype. [Ref. 8:pp. 7-8]

The Static Scheduler also utilizes PSDL as input. It schedules all components with real-time constraints after checking the compatibility of these timing constraints between operators.

The Dynamic Scheduler combines the time critical static schedule and the non-time critical components of the prototype into a piece of executable Ada® code called the dynamic schedule. This schedule is compiled, linked and finally executed. This execution is the test of the prototype and the design it represents.

d. Prototype Debugging and Modification.

The debugging and modification phase of the CAPS process actually takes place over the entire system and utilizes all the various tools. The debugging takes place in two places, first at the time of the execution of the static scheduler, at which time problems with timing constraints that would prohibit the production of the static schedule are identified and corrected, and secondly, during the execution of the dynamic schedule. At that time problems with the dynamic execution of the prototype are determined.

In order to resolve these problems, a modification mode is used in the user interface. This modification can go to any position in the defined prototype and modify specifications and control constraints. The user interface must interpret and make changes and, through the use of the design database, carry this change through the levels of decomposition. This could require new searches of the software base, deletion of existing operators or construction of new operators. If any changes are made, the modified PSDL program must be run through the ESS again. This process will be done iteratively until the prototype performs as desired or demonstrates problems in original requirements.

B. PRINCIPLES OF THE USER INTERFACE

The user interface of a complex software engineering system, such as CAPS, must do more than merely provide for data entry and display. It is also not enough that the interface simply provide a loose collection of software tools, rather it must provide sequence control, user guidance and data protection as well. A system that provides all these functions and unites the tools into a single system can be said to provide a software engineering environment.

1. Expert System and Sequence Control.

In order for software to provide all the functions of the user to system interface, it must rely on expert system technology. This expert system does not have to be based in an Artificial Intelligence language, rather the term expert system is used to imply an ability to guide the user through the desired processes. The interface must be able to interpret what the user is doing at any time and provide support. This expert system must communicate with the user to find out what they want to do at any moment when the system cannot be sure of the users intentions. This type of expert system is also referred to as "mutual consultation" [Ref. 9:p. 212].

While attempting to guide the user through a sequence of operations, it is important to remember to allow the user to remain in control. One method of achieving this goal is to allow the experienced user some flexibility of control. It is good to have an interface that can guide the user through the system, but an experienced user may regard such a system as "too restrictive". [Ref. 10:p. 48]

One of the benefits of a system that requires few control actions by the user is reduced memory load on the user. The user does not have to remember what to do next if the system can perform that function. Another way that this expert system can reduce the memory load on the user is to apply all known information to all possible uses. This frees the users from needing to remember what they have already done.

Four methods of sequence control are given in Figure 9 with their relative training requirements. A software engineering tool should limit the use of the dialogs that require high amounts of user training. This can not be avoided in a graphics tool, but limiting the number of commands the user needs to know can make the system easier to use. In a software engineering tool the interface should allow the user to concentrate on the developing software, rather than the developmental tool. This will increase the users productivity.

DIALOG TYPE	REQUIRED USER TRAINING
Question and answer	Little/none
Menu Selection	Little/none
Command Language	High
Graphic Interaction	High

Figure 9. Sequence Control Dialogs

The expert interface will insure that the user is placed in the proper tool in order to produce the needed prototype component. Additionally, this interface must "know" all the aspects of the developing software, so that the consistency of data entered in the different tools can be maintained. The syntax and semantics of

PSDL are an important part of this intelligent interface. This enables the interface to insure a valid PSDL prototype.

2. Data Entry.

Many of the same principles that applied to sequence control apply to data entry. Once again the system should try to limit the number of input actions required by the user and also reduce the memory load on the user.

In order to provide the most efficient form of data entry, the interface designers should consider textual, system driven and graphical inputs. Regardless of the method, there are some underlying qualities that any interface should provide.

Data should only be entered once and the system should be able to access this information wherever it is needed. The user should not be asked to reenter information that was already input.

Feedback should be provided during data entry. This includes displaying keyed entries character by character and giving an indication of mouse-down and mouse-up events in a graphics tool.

Data entry should be user paced. The system should not run away from a novice user, but at the same time it should be able to accommodate the speed of expert users.

3. Data Entry Methods.

The three data entry methods described above all have a place in the interface. It is important to choose the proper method for a particular task.

When a program is being developed the naming of elements and the description of arithmetic expressions are two tasks that seem best suited for textual input. When possible, this input should be done with the aid of a syntax directed editor such as the Cornell Program Synthesizer. This system allows the user to enter information without the frustrating syntactic details normally encountered. This can be a tremendous help when the user is not very familiar with the grammar of the programming language. [Ref. 11]

Graphical data entry is a very important area. With recent advances in graphic technology it is possible for the user to enter information in a pictorial form faster and more concisely than is possible in textual only environments. The top down refinement of a software system is a logical use of graphic editing. The decomposition from one level into many lower level components can be easily represented by the human mind in a data flow diagram. In general, it can be stated that people prefer pictures over words for describing structures, so graphical data entry is desirable for this purpose. [Ref. 12: p. 152]

The final method of interface is through the use of questions and response in a system driven interface. This system asks the user questions and takes different actions based on the response of the user. Although this method is more restrictive, it has its place in the user interface when the users cannot be given freedom to enter anything they want. At the cost of flexibility, the user can be guided to enter only information that is valid.

User interface design is more than just providing a method to get the information from user to machine and back. The overriding factor is keeping the user in control and this control should only be limited in situations where possible errors would greatly degrade the system.

III. DESIGN OF THE USER INTERFACE.

In Chapter II, CAPS was introduced and discussed as a group of individual components. In this chapter, the design of an interface that links these tools together into a single software engineering tool will be described. The design of the interface includes the specification of the inputs and outputs of the individual components as well as a complete description of the previously undefined design database. The interfaces involved in the the construction, execution and debugging/modification of the prototype will be described. However, a brief introduction to the Bourne Shell Scripts of the Unix[2] operating system is necessary.

A. BOURNE SHELL SCRIPTS AND UNIX OPERATING SYSTEM

The Bourne Shell Script is a way to perform a set of Unix commands contained in a single file called a script. The Bourne Shell provides string-valued variables, if-then-else logic, case statements, and for and while loops. In short, all the constructs required to control the operation of the system are available. Within the shell, all Unix commands can be utilized, thus providing a very powerful control environment [Ref. 13]. The Unix System allows for a collection of different tools, many written in different languages. Once the individual programs are compiled, they become executable Unix files and their source is transparent to the system. Writing the interface in the Bourne Shell allowed it to be on a higher level than the system components. An additional benefit of the Unix system is the ability to communicate with separate system components with the Unix argument and the

[2] UNIX is a trademark of AT&T Bell Laboratories.

shared Unix files. Any function required by the user interface could be performed in the Unix environment.

B. PROTOTYPE CONSTRUCTION

The Design Database (DDB), Software Base (SB), Graphic Editor (GE), Syntax Directed Editor (SDE), and the Rewrite System are utilized in the construction of the PSDL prototype. A description of the interface and function of each of these components as well as the function of the user interface in the coordination of the construction effort will be presented.

1. The Design Database.

The Design Database is a hierarchical storage structure for the development of the PSDL program. This structure is a multiway tree with each node containing:

- PSDL Specification part
- PSDL Implementation part (Graph or Ada®)
- Graphic Record (if implementation is Graph)
- PSDL Control Constraints part (if implementation is Graph).

The Specification part can be further divided to obtain the various elements of the specification. In particular, it can be broken into operator name, inputs, outputs, states, and MET individually. The Implementation part consists of the link statements produced in the Graphic Editor or written or retrieved Ada® code. The Graphic Record is the data used only by the Graphic Editor that is used to redraw the data flow diagram.

Each level of the tree is produced by the decomposition of the parent operator. The database is able to recognize the relation of parent and child. This allows queries of the type find child and find parent, as well as a search by operator name. Finally the DDB must be able to traverse the entire tree in a breadth first order to produce the PSDL program.

The DDB inputs are:

- Graphic Record (from GE)
- PSDL Implementation (Graph or Ada®)
- PSDL Control Constraints
- PSDL Specification
- Commands (from UI).

The DDB outputs are the same as the inputs with the addition of the complete PSDL program.

The following operations were designed to enable the DDB to aid in the construction and modification of the prototype.

- Create Root Node. This operation allows for the creation of a tree of operators in the database.
- Create Child Node. This operation creates a new node for information storage and sets the parent-child relationship between this new node and its parent.
- Store Property. This operation stores a PSDL part (Specification, Implementation or Control Constraints), subpart (Operator Name, Input List, Output List, State List or Maximum Execution Time), or Graphic record in the named node.
- Get Property. This operation retrieves these same properties from the DDB.
- Get Children. This operation returns the names of all the children of the named operator.
- Delete Node. This operator removes the named operator from the DDB. Because of the hierarchical nature of the DDB, this operation will effectively remove the entire subtree that is rooted at the named operator.
- Traverse Tree. This operation performs a breadth-first traversal of the DDB that collects the PSDL components into a single program.

2. The Graphic Editor

The Graphic Editor is a graphics tool for drawing data flow diagrams (DFD). It is the part of CAPS where most of the input of the prototype descroption is done. The decomposition of a PSDL Operator into lower level operators defines the actual creation of the new nodes in the tree structure. The names of all operators and data streams are entered here. The editor insures a valid decomposition by checking the consistency of inputs, outputs, states and maximum execution times. The GE can also show the DFD of the parent operator to aid the user in retaining the place of a single operator in the prototype [Ref. 14].

Inputs to the GE include operators name, input, output, and state lists and its maximum execution time. The outputs from the GE are the PSDL link statements and the Graphic Record. The operations performed by the GE include drawing operators data streams, inputs, and outputs showing parent DFD, and loading and storing the Graphic Record.

3. The Syntax Directed Editor.

The Syntax Directed Editor produces syntactically correct PSDL and syntax checks existing PSDL files. The SDE reads in and completes partial PSDL specifications and produces PSDL Control Constraints.

Input to the SDE is the partial PSDL specifications produced in the UI and its output is syntactically correct PSDL specification and control constraints.

4. The Software Base

The Software Base of reusable Ada® modules has two parts; a query module and a maintenance module. The maintenance module is involved with the creation and upkeep of the database. All records must be stored by PSDL specification so that they can later be searched by the same specification. Although this part of the system is not utilized in the construction of the prototype, it should enjoy the same interface as the query module. [Ref. 15]

The query module receives the PSDL specification part and returns zero, one or more Ada® modules that meet those specifications.

5. The User Interface.

The user interface has two main functions during the construction of the prototype; sequence control of the construction effort, and the insurance of continuity of the level-to-level decomposition of the operators. The sequence control is performed by utilizing the if-then-else logic of the Bourne Shell. The consistency of the decomposition is harder to achieve.

In the decomposition of an operator, a number of child operators are produced through the use of the data flow diagram. Although this decomposition can produce any number of new operators with any number of data streams between them, the inputs and outputs of the system of child operators must be exactly the same as those of the parent. The Graphic Editor can insure this by reading in an input and output list. The GE will not allow any other inputs or outputs and if all these inputs and outputs are not utilized the user will be notified that the decomposition is not valid. Additionally the Graphic Editor can check to ensure that the maximum execution time for any of the children does not exceed that of the parent. Finally, a state variable in a child must also exist in the DFD decomposition as a self loop or a internal connection between two operators.

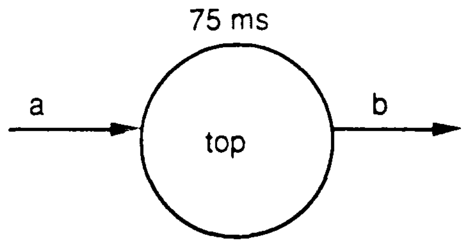
Figure 10 shows the decomposition of the operator *top* into its lower level components, *ape*, *bee*, *cat*, and *dog*. Figure 10(b) is a valid decomposition of the operator *top*, shown in Figure 10(a), because it has the same inputs and outputs (a and c). The GE would produce the link statements shown in Figure 10(c) for inclusion in the PSDL implementation.

As previously stated, the four operators *ape*, *bee*, *cat*, and *dog* represent the four child nodes of the node *top* in the DDB. These nodes are created, but the name of these operators is not the only thing known about them. Actually the link statements can be used to determine the inputs, outputs, names of any state variables, and maximum execution times of these operators. The User Interface reads the link statements and determines all of the information required to produce a partial specification. The specification has the names but not the types of the data streams. Production of this specification helps to ensure error free PSDL prototypes by relieving the user of the need to remember what he has previously entered and is in keeping with the guideline that data should be entered only once.

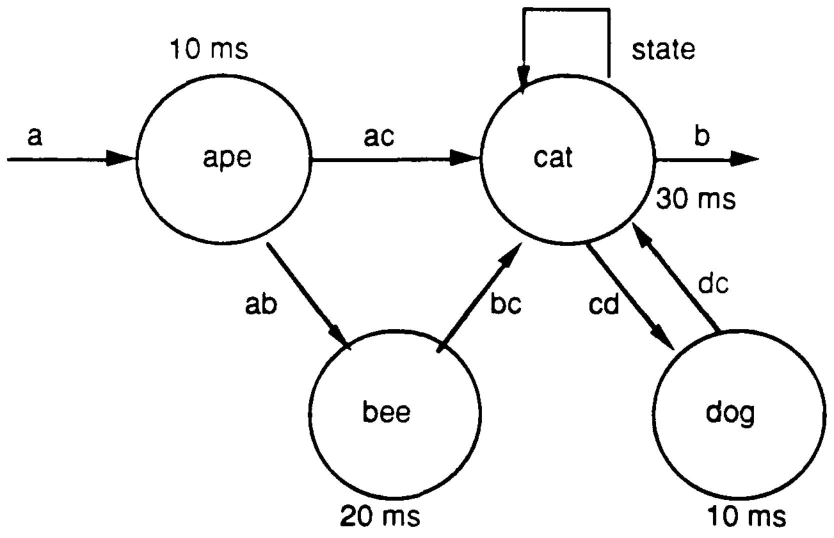
There is one additional place where the User Interface creates part of the PSDL program. The Implementation part of PSDL consists of link statements followed by a Data Stream List. This list consists of the internal data streams that were drawn in the DFD. The UI appends the DataStream List to the end of the link statements. To complete the Implementation part of the PSDL operator, the type of each of these data streams must be added in the SDE.

6. Sequence Control.

The construct module consists of a loop that continues while there are nodes in the DDB without an implementation part. The first incomplete node is found in the DDB and its specification is used to search the Software Base. If a match is found, that node is considered atomic and the Ada® code is placed in the implementation section of that operator. If there is no match the user is asked to



(a)



(b)

```
a.EXTERNAL-->ape
ab.ape:10ms-->bee
ac.ape:10ms-->cat
bc.bee:20ms-->cat
b.cat:30ms-->EXTERNAL
state.cat:30ms-->cat
cd.cat:30ms-->dog
dc.dog:10ms-->cat
```

(c)

Figure 10. Operator Decomposition

either decompose or write the Ada® implementation. If hand coding is done, this operator is again atomic and the Ada® code becomes the implementation part. Finally, if the user chooses to decompose the operator, the Graphic Editor draws the DFD is and produces the link statements. The User Interface reads these link statements and writes the partial specification for all newly created operators. New nodes in the DDB are created for each new operator. The Syntax directed editor is then called to complete the PSDL for the original composite operator.

The construct loop ends when all leaf nodes of the DDB are atomic. During the creation of a prototype, a rapid growth in the number of nodes is expected, as the high level operators are decomposed. Eventually the Ada® implementation for more of the lower level operators would be found in the SB and the growth of the tree would stop.

The construction process deals only with the production of operators. New data streams are produced in each operator. If these data streams are not atomic they must be defined in PSDL. All user defined data streams (ds) would appear outside the tree of operators on a level in the DDB equal to the root operator. Exceptions (ex) would also appear at this level, which is similar to a global type definition in Pascal. Figure 11 illustrates this structure. The tree of operators contains both composite operators (co) and atomic operators (ao).

C. PROTOTYPE EXECUTION

Prototype execution utilizes the Translator (TR), Static Scheduler (SS), and Dynamic Scheduler (DS) to produce an executable prototype in Ada®, that can test the design and requirements of the actual system.

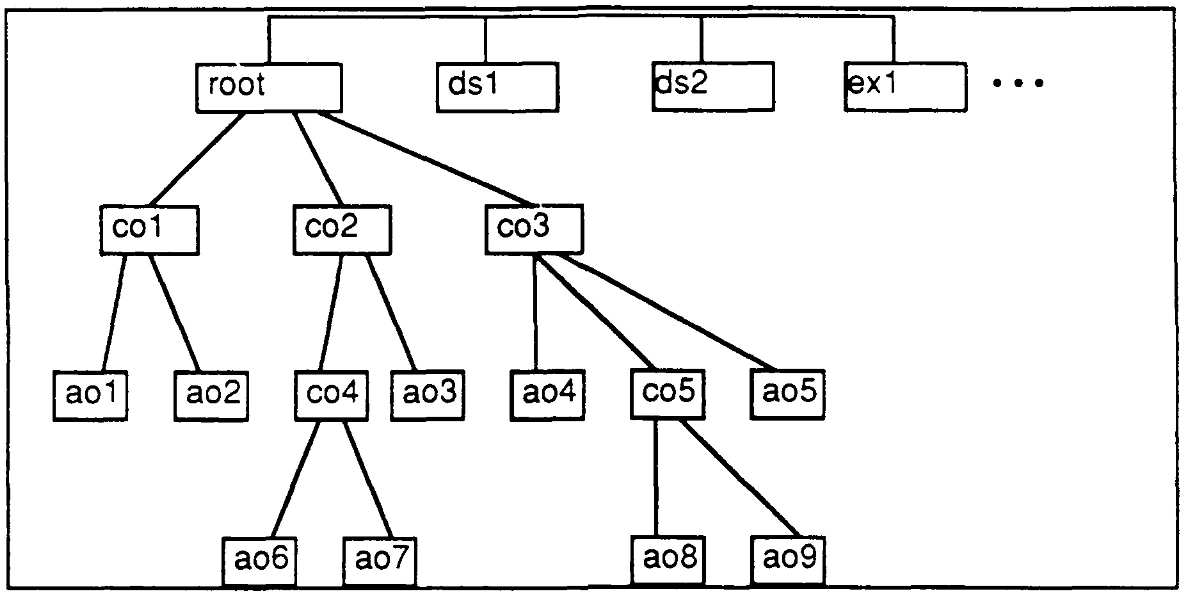


Figure 11. Conceptual DDB Structure

1. The Translator

The translator in CAPS translates the PSDL into Ada®. This is done by taking the Ada® implementation of the atomic operators and adding the control constraints of the composite operators to produce a group of loosely coupled Ada® modules. [Ref. 16]

The input to the translator is the PSDL program that was produced in the breadth first traversal of the DDB. The output is the package of Ada® modules.

2. The Static Scheduler

The Static Scheduler produces a schedule of time critical operators, if one can be produced. If it is impossible to produce a valid schedule because of the timing constraints set in the construction of the prototype, the user will be notified by the Debugger. This process will be described in the debugging and modification section.

3. The Dynamic Scheduler

The Dynamic Scheduler produces a dynamic version of the static schedule that includes a schedule of time critical operators, a collection of non critical operators and an exception handler that is the debugger. The Dynamic Scheduler adds the ability to run non-time critical operators in conjunction with the static schedule.

The produced dynamic schedule is a Ada® program that consists of two tasks and the exception handler. The higher priority task is the schedule of time critical operators. This task will execute until it reaches a designated milestone in the schedule, if it is ahead of schedule the secondary task, (non-time critical operators) will be executed for the amount of excess time. In the event that the prototype falls behind its time schedule at any milestone, an exception will be raised and the debugger will be started.

To aid in debugging, a trace and a graphical representation of the executing prototype are planned. The trace will list the name of the operator and the time that it is entered. This information is critical when the actual real-time performance of the prototype is being evaluated. The run time status of the prototype could be displayed by presenting the user with a tree that represents the nodes of the DDB. The node on the frontier of the tree that corresponds to the operator currently executing would be highlighted. This would give the user the ability to watch the actual execution of the prototype.

4. Sequence Control.

The Translator and Static Scheduler may be executed in any order, or simultaneously in a multitasking environment. After the Static Schedule is produced and a non-time critical operators identified, these operators must be grouped in a package for use in the Dynamic Scheduler. The Translator output is compiled and used in both the Static Schedule and the non-time critical package. These two packages then become part of the Dynamic Schedule, which must be compiled and linked before it is executed.

D. PROTOTYPE DEBUGGING AND MODIFICATION.

The debugging of the prototype takes place during the execution of the Static Scheduler, while the static schedule is being produced, and during the execution of the dynamic schedule of the prototype. The Debugger must be broken into two parts because exceptions caused by static problems arise before compilation, while many of the dynamic timing problems of a real-time system will not occur until the prototype has been compiled and is executing. [Ref. 17]

1. The Debugger.

The Debugger has two methods of correcting problems in the prototype. The first is through direct user interaction with the prototype and the second is through the Syntax Directed Editor and the Graphical Editor in the modification mode .

The Debugger gives the user a chance to make small changes to the prototype in the ESS. This allows rapid feedback as to the results of the change. The problem with this method of modification, is that these changes are temporary. The only way to make a permanent change to the prototype is with the SDE or GE.

2. Modifying the Prototypes.

There are many problems involved in implementing the facilities for the modification of a PSDL prototype. These problems stem from the fact that operators in the hierarchical structure of the DDB inherit information both up and down. In addition there are both graphical and textual views of an operator. These views actually hold different versions of the same information. A change in one view requires a change in the other.

a. Modifying an Operator.

If an operator is deleted the simple solution is to delete the entire subtree that has that operator as a root. This action is very severe and the DDB should record a historical version of the prototype at this time. If it is later shown that this deletion was an improper choice, this version of the prototype could be restored. A deletion would also require the modification of the DFD (and link statements) of the parent operator, where the deleted operator is first defined.

The addition of a new operator is as simple as a deletion. The new operator is added to the DDB tree and the construction mode of the user interface is entered. Construction continues until the new subtree is completely defined. In both deletetion and insertion the DFD of the parent operator must be modified to reflect the changes in its subtree.

The most significant problem in modifying an operator occurs when small changes in the specifications or control constraints of an operator are made. A change in the specification could cause changes in every node of its subtree. Equally as likely, a specification change could cause an atomic operator to become a complex operator if the search of the software base no longer yields a match. The search on modified specifications may yield a match that was not previously obtainable, therefor deleting a subtree and replacing it with an atomic operator.

This level-to-level consistency problem can move up the tree as well. Also a change at a child node may cause it to be different from the node required for the decomposition of the parent.

b. Consistency of Views.

A change of a textual component of PSDL may be carried over to the graphic representation. An example is the name of a data stream, that is changed in the implementation section of a PSDL operator, must be reflected in the link statement and also in the graphic record.

The graphical view of a change might be the best indication of the problems caused by deletions or changes. More effort is required in the area of prototype modification, if the same assurances of valid PSDL prototypes that are present in the construction mode are expected during modification.

E. TOP LEVEL USERS MANUAL

The top level users manual contains the four commands; *caps, construct, execute* and *modify*. This section describes these commands and the environment the user will be in when these commands are executed. The individual user manual for each system component, particularly the SDE and GE, should also be consulted.

1. The Caps Command.

The *caps* command is how the user initiates CAPS and it places the user in the user interface portion of CAPS. From this point the user can initiate one of the three remaining commands.

caps <new_proto_name>

The optional argument contains the name of a new prototype that is to be constructed. If the DDB is empty, this name will be used to create a new root node. If the argument is not used and the DDB is empty, the user will be asked to *Enter the root node name* . The response to this query will be used to create the root node. When the DDB is empty the user will always be placed in the construction mode as the execution and modification modes would not apply.

2. The Construct Command.

The *construct* command is used to place the user in the construction mode. In this mode the user is directed into the SDE and GE to create the PSDL program.

construct

This command will place the user in the location where the PSDL construction can begin or continue. The process is directed by the UI to insure the production of a complete and valid PSDL program. The particular aspects of both the SDE and GE users manuals should be reviewed in order to properly utilize these tools when they are called. The search of the Software Base, the storage and retrieval of components in the DDB, and the semantic checking of the UI are all transparent to the user.

The user is advised of the results of the software search and the completion of the construction with the below dialogs.

- *Software Search Complete - no match found.* This notifies the user that the search for an Ada® implementation for the given specification was unsuccessful. This would be followed by the question: *Do you want to decompose, y or n.* Based on the response the user will be placed in the GE or Ada® editor.
- *Software Search Complete - implementation found.* This indicates a successful retrieval of an Ada® implementation. The user is then asked to choose the next operator for implementation.
- *Select the next operator for implementation.* This dialog presents the user with a list of incomplete operators. The user then enters the name of the desired operator. This question will follow the completion of any implementation.
- *Construction Complete.* This message indicates the completion of the PSDL prototype. The user is then placed in the UI portion of CAPS where execution or modification can be selected.

3. **The Execute Command.**

The *execute* command places the user in the Execution Support System where the constructed prototype is executed to test the real-time performance.

execute

This command first checks for the existence of a completed prototype. If one does not exist the user is warned, *No Completed Prototype Available*, and placed back in the UI. When a complete prototype is available, the Translator, Static Scheduler and Dynamic Scheduler called in succession. The use of these components, as well as the Ada® compiler and linker, is transparent to the user. The user is informed of the status within the ESS with the below messages.

- *Translation Complete.*
- *Static Scheduler Complete.*
- *Dynamic Scheduler Complete.*
- *Compilation Complete.*
- *Linking Complete.*
- *Execution Complete.*

In the event of a problem in the scheduling or execution of the prototype, the user will be notified by the Debugger. The user has the option to make temporary corrections to the prototype in an attempt to achieve proper execution. All permanent changes must be made in the appropriate editor through the use of the *modify* command. The users manual for the Debugger should be consulted for the form of any commands, messages and dialogs.

4. The Modify Command.

The *modify* command is used to make changes to the prototype. The user is placed in the modification mode that insures that all changes are made consistently throughout the various levels in the DDB.

modify

This command asks the user to *Enter the name of the Operator to be modified.* The user must then call the SDE or GE to make the required change to the operator. The UI insures that the appropriate changes are made in the higher and lower levels of the DDB. The user will be asked to resolve the questions that will arise as these changes are carried out. Depending on the changes made the user might be required to enter the construction mode to complete the modified prototype. The design of this area of CAPS is not complete.

F. EVOLUTION OF THE USER INTERFACE.

The current design of CAPS is in an evolutionary stage. As CAPS changes the design of the User Interface must change to keep the system performing. At the present time, only a part of the total system has been implemented. As more of the functions of CAPS are implemented changes to the interface will be required.

IV. IMPLEMENTATION OF THE USER INTERFACE.

The user interface has not been completely implemented since all of the components of CAPS have not been implemented. The link statement analyzer of the UI has been implemented and is described here. The means of sequence control have been tested and proved to be feasible for use in the implementation of the interface. This chapter will outline the implementation of the link analyzer as well as a plan for the implementation for the remainder of the User Interface and the Dynamic Scheduler.

A. THE LINK STATEMENT ANALYZER.

The link statement analyzer is called *nodes* because it writes the partial specification for the newly created operators and is used to create new nodes in the DDB.

The program was written in Pascal, because that was the language most familiar to the author. Berkley Pascal was used because, like the remainder of the system, it can run on the Sun Workstation®[3] and the Unix Operating System.

The declaration section of *nodes.p* shown in Figure 12 describes the storage structure that is used. In this program the link statements are parsed and the operators are stored in the linked list of operators. Within a given operator a list of inputs, outputs, and states as well as the Maximum Execution Time are stored. State variables are determined by the existence of a data stream with the same operator as its two end points. Graphically this would appear as a self loop. There

[3] Sun Workstation® is a registered trademark of Sun Microsystems, Inc.

list of internal data streams collected for inclusion in the Implementation section of the parent operator. The storage structure is built up as each link statement is read in succession.

```
program CreateNodes (input,output);
const                                        (* Global Constants *)
  period = '.';
  colon = ':';
  arrow = '-';
  blank = ' ';
  EXTERNAL = 'EXTERNAL                                          ';
                                     (* 72 blanks *)
type
  string80 = packed array [0..79] of char;
  DataPtr = ^DataType;
  DataType = record                          (* Node for Linked List of Nodes *)
    Name: string80;
    Link: DataPtr;
    end; (* DataType *)
  OperPtr = ^Operator;
  Operator = record                          (* Node of Linked List of Opera ~rs *)
    OpName: string80;                        (* Operator Name *)
    InputList: DataPtr;                      (* Head Pointer to Input List *)
    InListTail: DataPtr;                     (* Tail Pointer to Input List *)
    OutputList: DataPtr;                     (* Head Pointer to Output List *)
    OutListTail: DataPtr;                    (* Tail Pointer to Output List *)
    StateList: DataPtr;                      (* Head Pointer to State List *)
    StateListTail: DataPtr;                  (* Tail Pointer to State List *)
    MET: string80;                           ( * Maximum Execution Time *)
    Link: OperPtr;
    end; (* Operator *)
var
  OpHead: OperPtr;                           (* Head of Operator List *)
  OpTail: OperPtr;                           (* Tail of Operator List *)
  DataHead: DataPtr;                         (* Head of Data List *)
  DataTail: DataPtr;                         (* Tail of Data List *)
```

Figure 12. Declaration Section of Nodes.p

After the link statements are read into the data structure, specification parts for each new node are written into dynamically created files. These files will be used to create new nodes in the DDB, with each file being the input for the create new node command. In addition to creating the child nodes, the Data Stream part of the PSDL implementation section for the parent node is produced. This data stream list consists of all new internal data streams created in the decompositions that are not state variables. The creation of both a child node and the Data Stream list in *nodes.p* are illustrated in Figure 13. This figure illustrates how the link statements that were produced in the GE (Figure 13(a)) are used by the UI to produce the partial specification of the newly defined operator, *ape, (Figure 13(b)).* The entire input and output of this example is shown in Appendix D. All of the semantic checking required to produce proper PSDL is included in the *nodes.p* program. *Nodes.p* is included as an important part of the construction subsystem.

```
ab.ape:10s-->bee
ac.ape:10s-->cat
bc.bee:20s-->cat
c.cat:30s-->EXTERNAL
state.cat:30s-->cat
cd.cat:30s-->dog
a.EXTERNAL-->ape
dc.dog:10s-->cat
```

(a)

```
NewNode.01

OPERATOR ape
SPECIFICATION
INPUT a
OUTPUT ab
       ac
MAXIMUM EXECUTION TIME 10s
```

(b)

Figure 13. Input and Output of Nodes.p

B. SEQUENCE CONTROL IN THE USER INTERFACE.

The Sequence Control of the construction and execution subsystems has been designed and tested. Complete implementation can be started when all the elements of CAPS are operating.

1. Implementation of the Construction Subsystem.

The goal of the construction subsystem is the production of the PSDL prototype. Before the construction process can begin the specification of the root operator must be written and the root node in the DDB created with the *createRootNode* command. This command to the DDB is executed by the UI when the user types the command *caps*. After this requirement is met the construction subsystem is entered.

The prototype construction is a loop that is executed until all the elements of the DDB are fully defined. The user interface maintains a list of nodes in a file called *node.list*. When new operators are created, their name is added to this list. As these operators are fully defined, they are removed from the list. When this *node.list* is empty the prototype is complete. The command: *while test-s node.list do* will perform this function in the Bourne Shell.

The first step in the construction is the completion of the PSDL specification. To do this the partial specification is retrieved from the DDB through the use of the *getProperty* command. This command is executed with the operators name on the first line in the file *ddb.in* and the name specification on the second. The DDB places the operator's specification in the file *ddb.out*. The SDE is invoked to complete and syntactically check the specification. This editor is called with the command *SDE ddb.out*. This specification is then sent to the Software Base by placing it in the file *psdl.spec* and then executing the file *Software_Base*.

response to the query is read from the file *SB_out*. If there is no match, the file is empty, otherwise the Ada® code is in this file.

The command *test-s SB_out* is used by the UI to determine if a match has been found. If it has, the Ada® code will be appended to the Implementation in the form of a PSDL comment. To do this, the words Ada® and the operator name are added followed by the retrieved Ada® code which is surrounded by the brackets of a PSDL comment.

If no match is found in the software search, the user is asked if they want to decompose the operator. A string-valued variable containing the users response is tested. When the response is negative, the user is placed in an editor, currently vi, where the Ada® implementation is written. The Ada® code is handled the same as the code that is retrieved from the Software Base. The implementation is stored in the DDB with the command *storeProperty* with the operator name followed by the Implementation section placed in *ddb.in*.

When the user wishes to decompose, the Graphic Editor is called and the data flow diagram and its corresponding link statements are produced. The GE is given the files *graph.name*, *graph.in*, *graph.out*, *graph.state* and *graph.met* which are made from data retrieved from the specification part of PSDL. The GE outputs *graph.link* and *graph.pic* are the link statements and graphical record respectively. *Graph.pic* is also an input, but it will be empty unless a DFD already exists.

The user interface creates *graph.name*, *graph.in*, *graph.out*, *graph.state*, and *graph.met* by getting the specification part of the node as described above, scanning for the information and writing it in the appropriate file. The output files *graph.pic* and *graph.link* are produced in the GE. The implementation section is stored in the DDB by writing the key words IMPLEMENTATION and GRAPH to the file *ddb.in* *ddb.in* and then concatenating the files *graph.link* and *psdl.ds* to it. The file *ddb.in* is then stored using the *storeProperty* command.

While reading the link statements to determine the internal data streams needed to produce *psdl.ds* the new nodes are identified. The program *nodes.p* dynamically produce files named *NewNode.01* to *NewNode.XX* depending on the number of new operators in the decomposition. The new nodes are created by executing the command *createChildNode* for each NewNode file. To execute *createChildNode*, *ddb.in* must have the name of the new node on the first line followed by the name of the parent node and then the partial specification produced by the link analyzer.

The loop is executed until all nodes are fully described with the leaf nodes being all atomic operators. The construction of the prototype is complete when the command *traverse* is executed. With the name of the root node in *ddb.in* , a breadth first traversal of the DDB is performed and the PSDL for each operator is concatenated to the file. This PSDL specification for the entire prototype is moved to the file *psdl* for use by the Execution Support System.

2. Implementation of the Execution Support System.

The implementation of the Execution Support System involves the coordination of the Translator, Static Scheduler, Dynamic Scheduler, and the Ada® compiler and linker as well.

To execute a prototype, the file psdl is first piped to the executable file *translator*. This program generates a package of loosely coupled Ada® components that each represent one of the leaf nodes of the prototype. This package is called *TL.a* and is used by both the Static and Dynamic Schedules.

The Static Schedule is produced by executing the attribute grammar preprocessor called *psdl_reader* with *psdl* piped as its input. The output file from *psdl_reader*, called *operator.info*, is the input to the next program, called *Static_Scheduler*. The *Static_Scheduler* produces a static scheduler, *SS.a*, as its output. A secondary output of the *Static_Scheduler* is the list of non-time critical operator *NTC.out*.

Both *SS.a* and *NTC.out* are used by the Dynamic Scheduler to produce an executable schedule called *DS.a*. This output schedule, which is written in Ada®, must be compiled and linked before execution. After schedule completion, the name of the executable prototype code is changed to be the same as the root operator.

3. Implementation of Debugging and Modification.

The debugger, as previously described, is broken into two parts; the Static Scheduler debugger and the executing prototype debugger. Both parts are embedded in the Execution Support System although it is not part of the execution of the prototype. If the debugger gets direct input from the user this information is used to try to keep the prototype executing. When the problems become too large the debugger writes the error message into the file *information* where the user can read it and attempt to modify the prototype.

The modification process is not completely defined at this time, but the DDB is equipped with all the operations that should be required. In addition to the operations that were previously discussed in the construction of the prototype, there are the *getParent* and *getChildren* commands for moving around the tree and the *deletNode* command for the removal of unneeded operators.

The *getParent* command is called with the known nodes name in *ddb.in*. The name of that nodes parent is placed in standard output and can be piped to wherever it is needed. The command *getChildren* is very similar except that the output is the list of child nodes and is placed in the file *ddb.out*.

The *deletNode* command simply deletes the node named in *ddb.in* and its entire subtree from the DDB. This command will require that a historical version of the DDB be saved. As the modification process is defined, additional operators in the various tools from the construction subsystem will be required.

C. EXAMPLE BOURNE SHELL SCRIPTS

The following example Bourne Shell scripts of the construct mode illustrate the commands that have been tested for use in the sequence control of the UI. These scripts are not complete, and have not been tested because the DDB was not implemented at the time of this writing.

1. The Construct Script.

The script for the construct mode is illustrated in Figure 14. This script is a loop that executes until the file *node.list* is empty, signifying a complete prototype. The first command in the loop calls the executable file *getChoice*. This file displays the list of incomplete operators to the user, who then selects the next operator for implementation. The choice is stored in the file *choice*. The command *cat choice spec >> ddb.in* places the name of the choice operator and the keyword SPECIFICATION, found in the file *spec*, in the file *ddb.in*. The command *getProperty* uses *ddb.in* to retrieve the specification of the operator from the DDB.

```
while test-s node.list do
      getChoice
      cat choice spec >> ddb.in
      getProperty
      SDE ddb.out
      mv ddb.out psdl.spec
      cat ddb.in psdl.spec >> temp
      mv temp ddb.in
      storeProperty
      Software_Base
      if test-s SB_out then
            echo "Software Search Complete - implementation found"
            addImpl
      else
            echo "Software Search Complete - no match found"
            echo "Do you want to decompose, y or n."
            read x
            if test $x = y then
                  GE
            else
                  adaEditor
                  addAdaImpl
            fi
      fi
done
echo "Construction Complete"
```

Figure 14. Construct Script

The output from the DDB, *ddb.out*, is sent to the SDE, where it is completed and syntax checked. The file is then moved with the *mv* command to the file *psdl.spec*, which is the input file for the Software Base. The next three commands create the file *temp* with the operator's name, the keyword SPECIFICATION and the completed specification in it. This file is moved to *ddb.in* where the *storeProperty* command stores it in the DDB.

The search of the SB is initiated by the command *Software_Base*. The results of the search are placed in the file *SB_out*. If there is no match this file is empty and it would fail the *test-s* command. If there is an implementation in the file, the user is notified with the message "*Software Search Complete - implementation found*", and the executable program *makeImpl* is called. This program creates a PSDL implementation section by placing the keywords IMPLEMENTATION and Ada in the file *ddb.in*. The operator name and the retrieved code, surrounded by the brackets ({}) of a PSDL comment are concatenated to complete *ddb.in*.

If no implementation is found in the SB, the user is notified with the command "*Software Search Complete - no match found*" and asked "*Do you want to decompose, y or n*". The user's response is read into the variable *x* if the response is "y" the GE is called. The script for the GE is demonstrated in the next section.

If the user does not want to decompose, the Ada® editor is called followed by *addAdaImpl* which adds the Ada® implementation to the PSDL operator just as *addImpl* does.

When node.list is empty, all the operators are completely defined, and the prototype is complete. The user is notified with the command " *Construction Complete*".

2. The Graphic Editor Script.

The GE script, shown in Figure 15, is embedded in the construct script. The primary purpose of the GE script is to get the inputs from the DDB, call the Graphical Editor, and then store the outputs back in the DDB.

The executable file *getInputs* calls the DDB command *getProperty* , with the proper keywords in the file *ddb.in,* to retrieve the operator name, inputs, outputs, states and maximum execution time from the operator specification, as well as the graphic record, if one was previously produced. This information is placed in the files *graph.name*, *graph.in*, *graph.out*, *graph.states*, *graph.met*, and *graph.pic* respectively. These files are the inputs to the GE.

The command *graph* puts the user in the editor where the DFD that defines the decomposition of the operator are produced. The outputs of the GE are the new graphic record *graph.pic* and the link statements in the file *graph.link*. *Graph.link* is piped to the link analyzer *nodes*, where the partial specification of the newly defined nodes and the data stream list are created.

```
get_inputs
graph
nodes < graph.links
cat graph.links psdl.ds >> psdl.imp
cat choice Imp Graph psdl.imp >> ddb.in
SDE ddb.in
storeProperty
cat  Graphic  graph.pic >> ddb.in
storeProperty
createNodes
```

Figure 15. Graphical Editor Script

The commands *Graph.link psdl.ds >> psdl.imp* and *cat choice Imp Graph psdl.imp >> ddb.in* produce the implementation of a composite operator. This is done by first combining the link statements and data stream list. The keywords IMPLEMENTATION, in the file *imp*, and GRAPH are added as well as the operator name. The PSDL implementation is then sent to the SDE where the user must add the types to the data streams in the data stream list. The user may also add any optional control constraints.

The *storeProperty* command is used to store both the implementation section and the graphic record. The *cat* command in between sets up the file *ddb.in* for the storage.

The last command in the script, *createNodes* takes the files of the form: *NewNode.XX*, that were created in the UI for each new operator defined in the the DFD, and creates a node in the DDB with the *createChildNode* command. The executable file *createNodes* must load the file *ddb.in* with the new operator name, parent operator name and the partia! specification, in *NewNode.XX*, for each new operator before executing the *createChildNode* command.

D. FUTURE IMPLEMENTATION.

There are two aspects to the future implementation of the User Interface. The first is implementation of the designed features of the system. As the individual components are completely implemented the UI will need to be expanded to provide all the features of the system to the user. After all the logical considerations of the interface are fully tested, the second part of the implementation can begin.

The second phase is more directly involved with the Human Factors Engineering of the system. Presently the major considerations are what the system does, what information is to be passed and which tool should be invoked next. The follow on work should be more involved with how the information is presented to and received from the user. The use of a more graphic interface with a response via the mouse, would be more compatible with the primarily graphic input of the system.

V. CONCLUSIONS

The goal of this thesis was to determine the requirements of a user interface for CAPS and to design such an interface. This expert interface will help transform CAPS from a collection of software engining tools, into a usable system for rapid prototyping.

A. FEASIBILITY ISSUES.

The work done in this thesis demonstrates that a user interface that supports the construction, execution and modification of an executable prototype can be produced. First the requirements of interface were defined. These included data entry, data protection, communication between elements and sequence control. A design of an interface that could support these requirements was then produced. Key aspects were tested and the implementation outlined.

This interface, along with the demonstration of the feasibility of most of the components of CAPS, has shown that the system can be built. The next step is to demonstrate that this prototyping effort can aide in the production of real-time embedded systems. It must be shown that CAPS can construct and execute these prototypes quickly with minimum user training.

B. BENEFITS OF THE STUDY

The primary benefit of the work done on the user interface is the advancement of CAPS. CAPS has demonstrated the potential for significant time and cost savings in the production of these real-time system. If the user interface can be powerful and friendly enough to promote the users interest in the system then CAPS will be more readily used.

A secondary benefit of this research is in the area of user interfaces for other systems. Other software engineering tools might be able to utilize some of the aspects of the CAPS user interface. A tool that uses both graphical and textual data entry and display could utilize the same type of control of data consistency between the two views. The use of the Bourne shell scripts might a be a useful model for some other system as well.

C. RECOMMENDATIONS.

There are three recommendations for the improvement of CAPS. These recommendations include the addition of two new areas of work to the system and a change in the primary method of data entry.

1. Primary Data Entry.

The original design of CAPS called for the majority of the data entry to be done in the SDE. As the system has been developed, the GE has proven to be the primary tool for the entry of new data. The DFD is the first place where the new operators and data streams are defined in PSDL. The UI already uses the information from the GE to produce the partial specification of the newly constructed operators and the data stream declaration. Currently the GE only names data streams and the type must be added in the SDE. The inclusion of types in the GE, and in the link statements, would eliminate the need to return to the SDE to complete the specifications and data stream lists.

2. Prototype Modification.

A method of prototype modification must be provided. A modification mode that not only allows changes to the prototype, but provides the same assurances of valid PSDL prototypes as the construction mode is required. Until this modification subsystem is in place, the rapid debugging of the prototype will not be possible.

3. Execution Monitoring.

The current design of CAPS does not include a means of monitoring the execution of the prototype. Some means of producing both a trace and a view of the execution of the prototype would greatly improve the ability to debug and verify the prototype's performance. The ability to trace a problem in the execution to one of the original requirements would aid in the validation or modification of these requirements.

D. APPLICATION OF CAPS IN THE DEPARTMENT OF DEFENSE AND THE DEPARTMENT OF THE NAVY.

The substantial investment in real-time embedded systems in DOD and DON points out the tremendous need for a tool such as CAPS. Since CAPS produces an executable prototype in Ada® that can be used to test the design of a software system before coding. MIL-H-48655B calls for requirements analysis, functional specification and verification before the actual coding of the system [Ref. 19]. Rapid Prototyping supports this objective. All embedded systems in DOD are to be written in Ada®, so it seems that CAPS is the tool that best supports the production of real-time embedded systems. Although CAPS is not yet a complete system, it does hold promise as an important tool for software engineering in DOD and DON.

There is much interest in the rapid development and acquisition of software for tactical systems in the Navy, as shown in OPNAVNOTE 5000. This instruction outlines a method for accelerated acquisition of tactical systems through the use of existing, "off the shelf" hardware and rapidly produced developmental software[Ref. 20]. The production of this software could be accelerated through the use of a software tool like CAPS.

Rapid prototyping has demonstrated a potential to be one possible solutions to the inherent problems of producing large real-time systems with traditional software engineering methodologies. CAPS shows promise as a practical software engineering tool.

APPENDIX A. PSDL GRAMMAR

```
Start = psdl
psdl = {component}
component = data_type | operator

data_type = "type"id type_spec type_impl
operator = "operator"id operator_spec operator_impl
type_spec = "specification" [type_decl] {"operator" id operator_spec}
            [functionality] "end"
type_impl = "implementation" "ada" id "{" text "}"
            | "implementation" type_name {"operator" id operator_impl} "end"
operator_spec = "specification" {interface} [functionality] "end"
operator_impl = "implementation" "ada" id "{" text "}"
              | "implementation" psdl_impl
type_decl = id_list ":" type_name {"," id_list ":" type_name}
functionality = [keywords] [informal_desc] [formal_desc]
psdl_impl = data_flow_diagram [streams] [timers] [control_constraints]
            [informal_desc] "end"
type_name = id "[" type_decl "]"
            | id
interface = attribute [reqmts_trace]
id_list = id {"," id }
keywords = "keywords" id_list
informal_desc = "description" "{" text "}"
formal_desc = "axioms" "{" text "}"
data_flow_diagram = "graph" {link}
streams = "data stream" type_decl
timers = "timer" id_list

attribute = generic_param
          | input
          | output
          | states
          | exceptions
          | timing_info
generic_param = "generic" type_decl
input = "input" type_decl
output = "output" type_decl
states = "states" type_decl "initially" expression_list
exceptions = "exceptions" id_list
```

```
timing_info = ["maximum execution time"  time]
              ["minimum calling period" time]
              ["maximum response time"  time]
reqmts_trace = "by requirements"  id_list
link = id  "."  id  [":" time]  "->"  id
control_constraints = "control constraints"  {constraint}
constraint = "operator"  id
              ["triggered"  [trigger] ["if" predicate] [reqmts_trace]]
              ["period" time [reqmts_trace]]
              ["finish within" time [reqmts_trace]]
              {constraint_options}

trigger = "by all"  id_list
          | "by some"  id_list
constraint_options = "output" id_list "if" predicate [reqmts_trace]
                     | "exception" id ["if" predicate] [reqmts_trace]
                     | timer_op id ["if" predicate] [reqmts_trace]
timer_op = "read timer"
           | "reset timer"
           | "start timer"
           | "stop timer"
expression_list = expression  {"," expression}
time = integer [unit]
unit =  "ms" | "sec" | "min" | "hours"
expression = constant
             | id
              | type_name "." id "(" expression_list ")"
predicate = relation {bool_op relation}
reiation = simple_expression
          | simple_expression  rel_op  simple_expression
simple_expression = [sign] integer [unit]
                    | [sign] real
                    | ["not"] id
                    | string
                     | ["not"] "(" predicate ")"
                     | ["not"] boolean_constant
bool_op = "and"  |  "or"
rel_op = "<"  |  "<="  |  ">"  |  ">="  |  "="  |  "/="  |  ":"
real = integer "." integer
integer = digit {digit}
boolean_constant =  "true"  |  "false"
numeric_constant = real  |  integer
constant = numeric_constant  |  boolean_constant
```

```
sign = "+" | "-"
char = any printable character except "}"
digit = "0 .. 9"
letter = "a .. z" | "A .. Z" | "_"
alpha_numeric = letter | digit
id = letter {alpha_numeric}
string = """ {char} """
text = {char}
```

APPENDIX B. BRAIN TUMOR TREATMENT SYSTEM EXAMPLE

```
OPERATOR Brain_tumor_treatment_system
SPECIFICATION
 INPUT patient_chart: medical_history,
          treatment_switch : boolean
 OUTPUT treatment_finished: boolean
 STATES temperature: real  INITIALLY 37.0
DESCRIPTION {This is an English description of the operator }
END
IMPLEMENTATION
GRAPH temperature.EXTERNAL--> hyperthermia_system
    patient_chart.EXTERNAL--> hyperthermia_system
    treatment_switch.EXTERNAL --> hyperthermia_system
    treatment_power.hyperthermia_system --> simulated_patient
    treatment_finished.hyperthermia_system --> EXTERNAL
    temperature.simulated_patient --> hyperthermia_system
DATA STREAM treatment_power : real
CONTROL CONSTRAINTS
 OPERATOR hyperthermia_system
  PERIOD 200 ms BY REQUIREMENTS shutdown
 OPERATOR simulated_patient
  PERIOD 200 ms
DESCRIPTION {some text about it}
END

OPERATOR hyperthermia_system
SPECIFICATION
 INPUT temperature:real,
         patient_chart:medical_history,
         treatment_switch: boolean
 OUTPUT treatment_power: real,
          treatment_finished: boolean
 MAXIMUM EXECUTION TIME 100 ms
  BY REQUIREMENTS temperature_tolerance
 MAXIMUM RESPONSE TIME 300 ms
  BY REQUIREMENTS shutdown
KEYWORDS medical_equipment,
      temperature_control,
      hyperthermia,
      brain_tumors
DESCRIPTION { }
END

IMPLEMENTATION
  GRAPH temperature.EXTERNAL --> start_up
       temperature.EXTERNAL --> maintain
       patient_chart.EXTERNAL --> start_up
       treatment_switch.EXTERNAL --> safety_control
       estimated_power.start_up --> safety_control
       estimated_power.maintain -->safety_control
       treatment_finished.maintain --> safety_control
       treatment_finished.start_up --> safety_control
       treatment_power.safety_control --> EXTERNAL

  DATA STREAM estimated_power: real
  TIMER treatment_time
  CONTROL CONSTRAINTS
    OPERATOR start_up
      TRIGGERED IF -42.4 > temperature
        BY REQUIREMENTS maximum_temperature
      STOP TIMER treatment_time
      RESET TIMER treatment_time IF temperature <= 37.0
    OPERATOR maintain
      TRIGGERED by all temperature, treatment_time
                        IF temperature >= 42.4
        BY REQUIREMENTS maximum_temperature
    OUTPUT treatment_finished IF treatment_time >= 45 min
      BY REQUIREMENTS treatment_time
      START TIMER treatment_time
        BY REQUIREMENTS treatment_time,
                temperature_tolerance
END

OPERATOR start_up
SPECIFICATION
  INPUT patient_chart: medical_history,
          temperature: real
  OUTPUT estimated_power: real,
          treatment_finished: boolean
  EXCEPTIONS err1, err2, err3
    BY REQUIREMENTS temperature_tolerance
```

```
DESCRIPTION { }
END
IMPLEMENTATION Ada start_up

OPERATOR maintain
SPECIFICATION
 INPUT temperature: real
 OUTPUT estimated_power: real,
            treatment_finished: boolean
 MAXIMUM EXECUTION TIME 90 ms
  BY REQUIREMENTS temperature_tolerance
 DESCRIPTION { }
END
IMPLEMENTATION Ada maintain
OPERATOR safety_control
SPECIFICATION
 INPUT treatment_switch,
          treatment_finished: boolean,
          estimated_power: real
 OUTPUT treatment_power: real
  BY REQUIREMENTS shutdown
 MAXIMUM EXECUTION TIME 10 ms
  BY REQUIREMENTS temperature_tolerance
 DESCRIPTION { }
END
IMPLEMENTATION Ada safety_control
OPERATOR simulated_patient
SPECIFICATION
  INPUT treatment_power : real
  OUTPUT temperature : real
  DESCRIPTION {  }
END
IMPLEMENTATION ADA simulated_patient
```

APPENDIX C. PASCAL PROGRAM nodes.p

```
program CreateNodes (input,output);
const                                           (* Global Constants *)
  period = '.';
  colon = ':';
  arrow = '-';
  blank = ' ';
  EXTERNAL = 'EXTERNAL                                                        ';
                                                (* 72 blanks *)
type
  string80 = packed array [0..79] of char;
  DataPtr = ^DataType;
  DataType = record                             (* Node for Linked List of Nodes *)
    Name: string80;
    Link: DataPtr;
    end; (* DataType *)
  OperPtr = ^Operator;
  Operator = record                             (* Node of Linked List of Operators *)
    OpName: string80;                           (* Operator Name *)
    InputList: DataPtr;                         (* Head Pointer to Input List *)
    InListTail: DataPtr;                        (* Tail Pointer to Input List *)
    OutputList: DataPtr;                        (* Head Pointer to Output List *)
    OutListTail: DataPtr;                       (* Tail Pointer to Output List *)
    StateList: DataPtr;                         (* Head Pointer to State List *)
    StateListTail: DataPtr;                     (* Tail Pointer to State List *)
    MET: string80;                              (* Maximum Execution Time *)
    Link: OperPtr;
    end; (* Operator *)
var
  OpHead: OperPtr;                              (* Head of Operator List *)
  OpTail: OperPtr;                              (* Tail of Operator List *)
  DataHead: DataPtr;                            (* Head of Data List *)
  DataTail: DataPtr;                            (* Tail of Data List *)

(*-------------------------------------------------*)
  procedure ReadToken(delimeter:char;
            var token:string80);
    (* Reads PSDL Link statements from standard input, one token at *)
    (* a time.  Delimeters are: period, colon, arrow and End of Line *)
  var
```

```
   ndx:integer;
   ch: char;
 begin
   ndx := 0;                           (* initialize *)
   read(ch);
   while (ch <> delimeter) and (not eoln) do
     begin
     token[ndx] := ch;                 (* Gets token character by character *)
     read(ch);                         (* until delimeter or eoln *)
     ndx := ndx + 1;
     end;  (* while*)
   if eoln then token[ndx] := ch;      (* Gets last character *)
   if delimeter = arrow then           (*before end of line *)
     begin
      read(ch); read(ch);              (* remove rest of arrow *)
     end;  (* if *)
   if eoln then readln;                (* resets line *)
 end;  (* ReadToken *)
(*-----------------------------------------------*)


 procedure ReadOperMet(var Oper1, Met: string80);
    (* Reads PSDL Link statements from standard input, one token at  *)
    (* a time. Determines Operator1 and Maximum Execution Time  *)
  var
   ndx:integer;
   ch: char;
  begin
   ndx := 0;                           (* initialize *)
   read(ch);
   while (ch <> colon) and (ch <> arrow) do
     begin
     Oper1[ndx] := ch;                 (* Gets token character by character *)
     read(ch);                         (* until delimeter or eoln *)
     ndx := ndx + 1;
     end;  (* while*)
   if ch = colon then                  (* end of line *)
     begin
     ndx := 0;
     read(ch);
     while ch <> arrow do
       begin
       Met[ndx] := ch;                 (* Gets token by character *)
```

```
        read(ch);                       (* until delimeter or eoln *)
        ndx := ndx + 1;
        end;  (* while*)
      end;  (* if *)
    read(ch); read(ch);                 (* remove rest of arrow *)
  end;  (* ReadOperMet*)
(*---------------------------------------------*)
  function OpSearch (Head: OperPtr;
             Target: string80): OperPtr;
    (* Searches Operator List for Target string, returns pointer *)
    (* to target if found, otherwise NIL *)
  begin
    if Head = nil then
      OpSearch := nil                   (* empty list *)
    else if Head^.OpName = Target then
         OpSearch := Head               (* target found *)
    else
         OpSearch := OpSearch(Head^.Link, Target);
  end;
(*---------------------------------------------*)
  procedure OpAdd (var Head: OperPtr;
              var Tail: OperPtr;
               Target: string80);
     (* Adds new Operator to end of linked list *)
  var
    p: OperPtr;                         (* temp pointer *)
  begin
    if Head = nil then                  (* List is empty *)
      begin
      new(p);                           (* Create new head node *)
      Head := p;
      Tail := p;
      p^.OpName := Target;              (* Initialize new list *)
      p^.InputList := nil;
      p^.InListTail := nil;
      p^.OutputList := nil;
      p^.OutListTail := nil;
      p^.Link := nil;
      end  (* if *)
    else                                (* List not empty *)
      begin
      new(p);                           (* Add new node after tail *)
      Tail^.Link := p;
    Tail := Tail^.Link;
    p^.OpName := Target;                (* Initialize new lists *)
    p^.InputList := nil;
    p^.InListTail := nil;
    p^.OutputList := nil;
    p^.OutListTail := nil;
    p^.StateList := nil;
    p^.StateListTail := nil;
    p^.Link := nil;
    end (* else *)
  end;  (* OpAdd *)

(*--------------------------------------------------*)

  function Search (Head: DataPtr;
             Target: string80): DataPtr;
     (* Searches Data Listfor Target string, returns pointer  *)
     (* to target if found, otherwise NIL *)
  begin
   if Head = nil then
     Search := nil                     (* empty list *)
   else if Head^.Name = Target then
       Search := Head                  (* target found *)
   else
        Search := Search(Head^.Link, Target);
  end;  (* Search *)
(*-----------------------------------------------*)
  procedure Add (var Head: DataPtr;
             var Tail: DataPtr;
               Target: string80);
   (* Adds new Data to end of linked lists *)
  var
   p: DataPtr;                         (* Temp pointer *)
  begin
   if Head = nil then                  (* List is empty *)
     begin
     new(p);                           (* Create new node *)
     Head := p;
     Tail := p;
     p^.Name := Target;                     (* Initialize new lists *)
     p^.Link := nil;
       end   (* if *)
   else                                     (* List not empty *)
```

```
    begin
    new(p);                                    (* Add new node after tail *)
    Tail^.Link := p;
    Tail := Tail^.Link;
    p^.Name := Target;                              (* Initialize new lists *)
    p^.Link := nil;
    end (* else *)
 end;  (* OpAdd *)
(*-----------------------------------------------*)
 procedure LoadDataStructure(var OpHead, OpTail: OperPtr;
                   var DataHead, DataTail: DataPtr);
  (* Loads tokens into Data Structures *)
 var
  Current: OperPtr;                             (* Temp pointer *)
  Data, Met:  string80;                         (* PSDL Tokens *)
  Oper1, Oper2:  string80;
 begin
  Data := blank;                           (* Initialize Strings *)
  Oper1 := blank;
  Met := blank;
  Oper2 := blank;
  while not eof do
   begin                                        (* Get tokens *)
    ReadToken(period,Data);
    ReadOperMet(Oper1,Met);
    ReadToken(' ',Oper2);
    if Oper1 <> EXTERNAL then          (* Keyword EXTERNAL is not *)
     begin                             (* an Operator *)
      Current := OpSearch(OpHead,Oper1);
      if Current = nil then
        begin      (* Add Operator 1 *)
        OpAdd(OpHead,OpTail,Oper1);
        Current := OpSearch(OpHead,Oper1);
       end;  (* if *)
      Current^.MET := Met;             (* Enter Maximun Execution Time *)
           (* Add Data to Operators Output List *)
      if Oper1 = Oper2 then
       begin
        if Search(Current^.StateList,Data) = nil then
          Add(Current^.StateList,Current^.StateListTail,
           Data);
       end
     else
```

```
          if Search(Current^.OutputList,Data) = nil then
            Add(Current^.OutputList,Current^.OutListTail,
             Data);
        end;  (* if *)
      if Oper2 <> EXTERNAL then          (* Keyword EXTERNAL is not *)
        begin                            (* an Operator *)
         Current := OpSearch(OpHead,Oper2);
         if Current = nil then
           begin              (* Add Operator 2 *)
            OpAdd(OpHead,OpTail,Oper2);
            Current := OpSearch(OpHead,Oper2);
           end;  (* if *)
                  (* Add Data to Operators Input List *)
         if Oper1 = Oper2 then
          begin
           if Search(Current^.StateList,Data) = nil then
             Add(Current^.StateList,Current^.StateListTail,
              Data);
          end
        else
           if Search(Current^.InputList,Data) = nil then
             Add(Current^.InputList,Current^.InListTail,
              Data);
        end;  (* if *)
                  (* Enter new internal Data Streams in Data List *)
      if ((Oper1 <> EXTERNAL) and (Oper2 <> EXTERNAL)) and
         (Oper1 <> Oper2) then
        if Search(DataHead,Data) = nil then
          Add(DataHead,DataTail,Data);
      Data := blank;                     (* Reset Strings *)
      Oper1 := blank;
      Met := blank;
      Oper2 := blank;
      end  (* while *)
  end;  (* LoadDataStructure *)
(*--------------------------------------------------*)
  procedure WriteString(var File:text; Str: string80);
  var
    ndx: integer;
  begin
    ndx := 0;
    while Str[ndx] <> ' ' do
      begin
    write(File,Str[ndx]);
    ndx := ndx + 1;
    end;  (* while *)
  end;  (* WriteString *)
(*-------------------------------------------------*)
  procedure MakePSDL(Head: OperPtr);
   (* Generates partial PSDL Specification for each new Operator *)
   (* in the Graphical decomposition *)
  type
   string10 = packed array [0..9] of char;
  var
   Current: OperPtr;                    (* Temp pointers *)
   InTemp:  DataPtr;
   OutTemp: DataPtr;
   StateTemp: DataPtr;
   OutFile: text;
   NodeName: string10;             (* Unix file name *)
  begin
   Current := Head;
   NodeName := 'NewNode.01';            (* First file name *)
   while Current<> nil do
     begin
      rewrite(OutFile,NodeName);        (* Create new file *)
              (* output PSDL *)
      write(OutFile,'OPERATOR ');
      WriteString(OutFile,Current^.OpName);
      writeln(OutFile);
      writeln(OutFile);
      writeln(OutFile,'SPECIFICATION');
      writeln(OutFile);
      InTemp := Current^.InputList;
      if InTemp <> nil then             (* Generate Input list *)
        begin
         write(OutFile,'INPUT ');
         WriteString(OutFile,InTemp^.Name);
         writeln(OutFile);
         InTemp := InTemp^.Link;
         while InTemp <> nil do
           begin
            write(OutFile,'     ');
            WriteString(OutFile,InTemp^.Name);
            writeln(OutFile);
            InTemp := InTemp^.Link;
    end;  (* while *)
   writeln(OutFile);
  end;  (* if *)
OutTemp := Current^.OutputList;
if OutTemp <> nil then
  begin                              (* Generate Output list *)
   write(OutFile,'OUTPUT ');
   WriteString(OutFile,OutTemp^.Name);
   writeln(OutFile);
   OutTemp := OutTemp^.Link;
   while OutTemp <> nil do
    begin
     write(OutFile,'     ');
     WriteString(OutFile,OutTemp^.Name);
     writeln(OutFile);
     OutTemp := OutTemp^.Link;
    end;  (* while *)
   writeln(OutFile);
  end;  (* if *)
StateTemp := Current^.StateList;
if StateTemp <> nil then
  begin      (* Generate State list *)
   write(OutFile,'STATE ');
   WriteString(OutFile,StateTemp^.Name);
   writeln(OutFile);
   StateTemp := StateTemp^.Link;
   while StateTemp <> nil do
    begin
     write(OutFile,'     ');
     WriteString(OutFile,StateTemp^.Name);
     writeln(OutFile);
     StateTemp := StateTemp^.Link;
    end;  (* while *)
   writeln(OutFile);
  end;  (* if *)
write(OutFile,'MAXIMUM EXECUTION TIME ');
WriteString(OutFile,Current^.MET);
writeln(OutFile);
writeln(OutFile);
Current := Current^.Link;
          (* Dynamically create new file name *)
if NodeName[9] = '9' then
  begin
```

```
      NodeName[9] := '0';
      NodeName[8] := succ(NodeName[8]);
      end  (* if *)
    else
      NodeName[9] := succ(NodeName[9]);
    end;  (* while *)
  end;  (* MakePSDL *)
(*------------------------------------------*)
procedure MakeDataStream (Head: DataPtr);
  (* Generate PSDL Data Stream *)
var
  Temp: DataPtr;
  Outfile: text;
begin
  rewrite(Outfile,'psdl.ds');
  writeln(Outfile);
  if Head <> nil then
    begin
    Temp := Head;
    write(Outfile,'DATA STREAM ');
    WriteString(Outfile,Temp^.Name);
    writeln(Outfile);
    Temp := Temp^.Link;
      while Temp <> nil do
        begin
        write(Outfile,'            ');
        WriteString(Outfile,Temp^.Name);
        writeln(Outfile);
        Temp := Temp^.Link;
        end;  (* while *)
    writeln(Outfile);
    end;  (* if *)
  end;  (* MakeDataStream *)
(*--------------------------------------------------*)
begin    (* main *)
  LoadDataStructure(OpHead, OpTail, DataHead, DataTail);
  MakePSDL(OpHead);
  MakeDataStream(DataHead);
end.      (* main *)
```

APPENDIX D. INPUT AND OUTPUT LINK ANALYZER

```
LINK STATEMENTS
ab.ape:10s-->bee
ac.ape:10s-->cat
bc.bee:20s-->cat
b.cat:30s-->EXTERNAL
state.cat:30s-->cat
cd.cat:30s-->dog
a.EXTERNAL-->ape
dc.dog:10s-->cat


NEWLY CREATED NODES

NewNode.01:
OPERATOR ape
SPECIFICATION
INPUT a
OUTPUT ab
        ac
MAXIMUM EXECUTION TIME 10s

NewNode.02:
OPERATOR bee
SPECIFICATION
INPUT ab
OUTPUT bc
MAXIMUM EXECUTION TIME 20s

NewNode.03:
OPERATOR cat
SPECIFICATION
INPUT ac
      bc
      dc
OUTPUT b
        cd
STATE state
MAXIMUM EXECUTION TIME 30
```

```
NewNode.04:
OPERATOR dog
SPECIFICATION
INPUT cd
OUTPUT dc
MAXIMUM EXECUTION TIME 10s
```

```
Data Stream part for OPERATOR top:
DATA STREAM ab
            ac
            bc
            cd
            dc
```

APPENDIX E. BOURNE SHELL SCRIPTS

CONSTRUCT SCRIPT:

```
while test-s node.list do
      getChoice
      cat choice spec >> ddb.in
      getProperty
      SDE ddb.out
      mv ddb.out psdl.spec
      cat ddb.in psdl.spec >> temp
      mv temp ddb.in
      storeProperty
      Software_Base
      if test-s SB_out then
            echo "Software Search Complete - implementation found"
            addImpl
      else
            echo "Software Search Complete - no match found"
            echo "Do you want to decompose, y or n."
            read x
            if test $x = y then
                  GE
            else
                  adaEditor
                  addAdaImpl
            fi
      fi
done
echo "Construction Complete"
```

GRAPHIC EDITOR SCRIPT:

```
get_inputs
graph
nodes < graph.links
cat graph.links psdl.ds >> psdl.imp
cat choice Imp Graph psdl.imp >> ddb.in
SDE ddb.in
```

```
storeProperty
cat Graphic graph.pic >> ddb.in
storeProperty
createNodes
```

REFERENCES

1. Booch, G., *Software Engineering with Ada®, 2nd ed.*, Benjamin Cummings Publishing Co., Inc., 1987.

2. Pressman, R., *Software Engineering: A Beginners Guide*, pp. 1-93, McGraw-Hill Book Company, 1988.

3. Luqi, *Rapid Prototyping for Large Software System Designs,* Doctoral Dissertation, University of Minnesota, Duluth, Minnesota, 1986.

4. Luqi, Berzins, V., "Rapidly Prototying Real-Time Systems," *IEEE Software*, v. 5, pp. 25-36, September 1988.

5. Luqi, Ketabchi, M., "A Computer Aided Protyping System," *IEEE Software*, v. 5, pp. 66-72, March 1988.

6. Janson, D., *A Static Scheduler for the Computer Aided Prototyping System: An Implementation Guide*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1988.

7. O'Hern, T., *A Conceptual Level Design for a Static Scheduler for Hard Real-Time Systems*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1988.

8. Naval Postgraduate School, Technical Report NPS52-88-009, *Automated Translation from a Prototyping Language into Ada®,* by C. Moffitt, II, and Luqi, pp. 7-10, April 1988.

9. The Mitre Corporation, *Guidelines for Designing User Interface Software*, U.S. Department of Commerce, National Technical Information Service, August 1986.

10. Dumas, J., *Designing User Interfaces for Software*, Prentice-Hall, 1988.

11. Teitelbaum, T., and Reps, T., "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Connunications of the ACM*, v. 24:9, pp. 563-573, September 1981.

12. Shi-Kuo, C., and others, *Visual Languages*, pp. 155-157, Plenum Press, 1986.

13. Sun Microsystems, *Doing More with UNIX: Beginner's Guide*, pp. 123-149, 1986.

14. Thorstenson, R., *A Graphical Editor for the Computer Aided Prototyping System (CAPS)*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

15. Galik, D., *A Conceptual Design of a Software Base Management System for the Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

16 Altizer, C., *Implementation of a Language Translator for a Computer Aided Prototyping System*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

17. Wood, M., *An Execution Support System for Rapid Prototyping*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

18. Marlowe, L., *A Scheduler for Critical Timing Constraints*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

19. Department of Defense, Military Specification MIL-H-48655B, *Human Engineering Requirements for Military Systems, Equipment and Facilities*, 31 January 1979.

20. Department of the Navy, OPNAV NOTICE 5000, *Navy Rapid Prototyping and the Research Development and Acquistion Process,* 20 February 1988.

BIBLIOGRAPHY

Barstow, D., and others, *Interactive Programming Environments*, McGraw-Hill Book Company, 1984.

Department of Defense, Military Standard MIL-STD-1472C, *Human Engineering Design Criteria for Military Systems, Equipment and Facilities*, 1 September 1983.

McDermid, J., *Intergrated Project Support Environments*, Peter Peregrinus Ltd., 1985.

Porter, S., *A Design of a Syntax-Directed Editor for CAPS*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988.

Sanders, M., and McCormick, E., *Human Factors in Engineering and Design*, McGraw-Hill Book Company, 1987.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Commandant of the Marine Corps Code TE 06 Headquarters, U.S. Marine Corps Washington, DC 20380-0001	1
4.	Commanding General Marine Corps Research, Development and Acquistion Command (Code C2A) Attn. CAPT H.G. Raum, USMC Washington, DC 20380-0001	1
5.	Office of Naval Research Office of the Chief of Naval Research Attn. CDR Michael Gehl, Code 1224 800 N. Quincy Street Arlington, VA 22217-5000	1
6.	Space and Naval Warfare Systems Command Attn. Dr. Knudsen, Code PD 50 Washington, DC 20363-5100	1
7.	Ada Joint Program Office OUSDRE(R&AT) Pentagon Washington, DC 20301	1
8.	Naval Sea Systems Command Attn. CAPT Joel Crandall National Center #2, Suite 7N06 Washington, DC 22202	1

9.	Office of the Secretary of Defense Attn. CDR Barber STARS Program Office Washington, DC 20301	1
10.	Office of the Secretary of Defense Attn. Mr. Joel Trimble STARS Program Office Washington, DC 20301	1
11.	Commanding Officer Naval Research Laboratory Code 5150 Attn. Dr. Elizabeth Wald Washington, DC 20375-5000	1
12.	Navy Ocean System Center Attn. Linwood Sutton, Code 423 San Diego, California 92152-500	1
13.	National Science Foundation Attn. Dr. William Wulf Washington, DC 20550	1
14.	National Science Foundation Division of Computer and Computation Research Attn. Dr. Peter Freeman Washington, DC 20550	1
15.	National Science Foundation Director, PYI Program Attn. Dr. C. Tan Washington, DC 20550	1
16.	Office of Naval Research Computer Science Division, Code 1133 Attn. Dr. Van Tilborg 800 N. Quincy Street Arlington, VA 22217-5000	1

17. Office of Naval Research 1
Applied Mathematics and Computer Science, Code 1211
Attn. Mr J. Smith
800 N. Quincy Street
Arlington, VA 22217-5000

18. Defense Advanced Research Projects Agency (DARPA) 1
Integrated Strategic Technology Office (ISTO)
Attn. MAJ Mark Pullen, USAF
1400 Wilson Boulevard
Arlington, VA 22209-2308

19. Defense Advanced Research Projects Agency (DARPA) 1
Director, Naval Technology Office
1400 Wilson Boulevard
Arlington, VA 2209-2308

20. National Science Foundation 1
Director of Computer and Computation Research
Attn. Dr. Tom Keenan
Washington, DC 20550

21. Defense Advanced Research Projects Agency (DARPA) 1
Director, Strategic Technology Office
1400 Wilson Boulevard
Arlington, VA 2209-2308

22. Defense Advanced Research Projects Agency (DARPA) 1
Director, Prototype Projects Office
1400 Wilson Boulevard
Arlington, VA 2209-2308

23. Defense Advanced Research Projects Agency (DARPA) 1
Director, Tactical Technology Office
1400 Wilson Boulevard
Arlington, VA 2209-2308

24. COL C. Cox, USAF 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, DC 20318-8000

25. LTCOL Kirk Lewis, USA 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, DC 20318-8000

26. U.S. Air Force Systems Command 1
Rome Air Development Center
RADC/COEAttn. Mr. Samuel A. DiNitto, Jr.
Griffis Air Force Base, NY 13441-5700

27. U.S. Air Force Systems Command 1
Rome Air Development Center
RADC/COE
Attn. Mr. William E. Rzepka
Griffis Air Force Base, NY 13441-5700

28. Professor Luqi 1
Code 52LQ
Naval Postgraduate School
Computer Science Department
Monterey, CA 93943-5000